

Properties

LaunchSettings.json

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5288",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7124;http://localhost:5288",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

API

ApiExceptionHandler.cs

```
using BridgeControl.Api.Exceptions;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Api;

/// <summary>
/// Converts expected database conflicts and unexpected failures into safe API error responses.
/// </summary>
public sealed class ApiExceptionHandler(IProblemDetailsService problems,
ILogger<ApiExceptionHandler> logger) : IExceptionHandler
{
    /// <summary>
    /// Converts handled exceptions into the API's problem-details response.
    /// </summary>
}
```

```

    public async ValueTask<bool> TryHandleAsync(HttpContext context, Exception exception,
CancellationToken cancellationToken)
    {
        ProblemDetails problem = exception switch
        {
            LicenseValidationException invalid => new()
            {
                Status = 400,
                Title = "Invalid license.",
                Detail = invalid.Message
            },
            AgentRateLimitException => new()
            {
                Status = 429,
                Title = "Too many requests."
            },
            AgentValidationException invalid => new()
            {
                Status = 400,
                Title = "Invalid Agent request.",
                Detail = invalid.Message
            },
            ConfigurationValidationException invalid => new()
            {
                Status = 400,
                Title = "Invalid configuration.",
                Detail = invalid.Message
            },
            ArtifactValidationException invalid => new()
            {
                Status = 400,
                Title = "Invalid artifact.",
                Detail = invalid.Message
            },
            ApiAccessException access => new()
            {
                Status = access.StatusCode,
                Title = access.StatusCode switch
                {
                    401 => "Authentication failed.",
                    403 => "Permission denied.",
                    _ => "Resource not found."
                }
            },
            ApiConflictException conflict => new()
            {
                Status = StatusCodes.Status409Conflict,
                Title = "The operation conflicts with the current state.",
                Type =
                    $"https://bridgecontrol/errors/{Uri.EscapeDataString(conflict.Code)}",
                Detail = conflict.Message
            },
            DbUpdateConcurrencyException => new()
            {
                Status = StatusCodes.Status409Conflict,
                Title = "The resource has changed. Reload it before trying again.",
                Type = "https://bridgecontrol/errors/concurrency-conflict"
            },
            DbUpdateException { InnerException: SqlException { Number: 2601 or 2627 or
547 } } => new()
            {
                Status = StatusCodes.Status409Conflict,

```

```

        Title = "The change conflicts with existing or referenced data.",
        Type = "https://bridgecontrol/errors/data-conflict"
    },
    _ => new()
    {
        Status = StatusCodes.Status500InternalServerError,
        Title = "An unexpected error occurred.",
        Type = "https://bridgecontrol/errors/internal-error"
    }
};
if (problem.Status == StatusCodes.Status500InternalServerError)
{
    // Exception messages can contain SQL values or credentials; correlate by
    type and request ID.
    logger.LogError("API request {TraceId} failed with {ExceptionType}.",
context.TraceIdentifier, exception.GetType().Name);
}
context.Response.StatusCode = problem.Status!.Value;
if (problem.Status == 429) { context.Response.Headers.RetryAfter = "60"; }
ApiServiceRegistration.CustomizeProblem(context, problem);
if (!await problems.TryWriteAsync(new ProblemDetailsContext { HttpContext =
context, ProblemDetails = problem }))
{
    await context.Response.WriteAsJsonAsync(problem, options:
(System.Text.Json.JsonSerializerOptions?)null,
        contentType: "application/problem+json", cancellationToken:
cancellationToken);
}
return true;
}
}

```

ApiJson.cs

```

using System.ComponentModel.DataAnnotations;
using System.Globalization;
using System.Text.Json;
using System.Text.Json.Serialization;
using System.Text.Json.Serialization.Metadata;

namespace BridgeControl.Api;

/// <summary>
/// Applies the public JSON contract while preserving credentials and configuration JSON values.
/// </summary>
public static class ApiJson
{
    /// <summary>
    /// Registers the API string and UTC timestamp serialization conventions.
    /// </summary>
    public static void Configure(JsonSerializerOptions options)
    {
        options.PropertyNamingPolicy = JsonNamingPolicy.CamelCase;
        options.Converters.Add(new JsonStringEnumConverter(allowIntegerValues: false));
        options.Converters.Add(new UtcDateTimeOffsetConverter());
        DefaultJsonTypeInfoResolver resolver = new();
        resolver.Modifiers.Add(type =>
        {
            foreach (JsonPropertyInfo property in type.Properties)
            {

```

```

        if (property.PropertyType == typeof(string)
            &&
property.AttributeProvider?.IsDefined(typeof(PreserveWhitespaceAttribute), true) != true)
        {
            property.CustomConverter = new InputStringConverter(

property.AttributeProvider?.IsDefined(typeof(EmailAddressAttribute), true) == true);
        }
    }
});
options.TypeInfoResolver = resolver;
}

/// <summary>
/// Normalizes incoming JSON strings using the API's input conventions.
/// </summary>
private sealed class InputStringConverter(bool email) : JsonConverter<string>
{
    /// <summary>
    /// Reads and normalizes an input JSON string.
    /// </summary>
    public override string? Read(ref Utf8JsonReader reader, Type type,
JsonSerializerOptions options)
    {
        string? value = reader.GetString()?.Trim();
        return email ? value?.ToLowerInvariant() : value;
    }

    /// <summary>
    /// Writes a string as a JSON value.
    /// </summary>
    public override void Write(Utf8JsonWriter writer, string value,
JsonSerializerOptions options)
        => writer.WriteStringValue(value);
}

/// <summary>
/// Enforces the API's UTC timestamp serialization conventions.
/// </summary>
private sealed class UtcDateTimeOffsetConverter : JsonConverter<DateTimeOffset>
{
    /// <summary>
    /// Reads a timestamp and enforces the API UTC representation.
    /// </summary>
    public override DateTimeOffset Read(ref Utf8JsonReader reader, Type type,
JsonSerializerOptions options)
        => reader.GetDateTimeOffset().ToUniversalTime();

    /// <summary>
    /// Writes a timestamp using the API UTC representation.
    /// </summary>
    public override void Write(Utf8JsonWriter writer, DateTimeOffset value,
JsonSerializerOptions options)
        => writer.WriteStringValue(value.UtcDateTime.ToString("O",
CultureInfo.InvariantCulture));
}
}

```

ApiServiceRegistration.cs


```

using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ModelBinding.Metadata;

namespace BridgeControl.Api;

/// <summary>
/// Registers shared JSON, validation and ProblemDetails conventions for all controllers.
/// </summary>
public static class ApiServiceRegistration
{
    /// <summary>
    /// Registers shared API validation, serialization, and error-response conventions.
    /// </summary>
    public static IMvcBuilder AddApiConventions(this IServiceCollection services)
    {
        services.AddProblemDetails(options => options.CustomizeProblemDetails = context =>
            CustomizeProblem(context.HttpContext, context.ProblemDetails));
        services.AddExceptionHandler<ApiExceptionHandler>();
        return services.AddControllers(options =>
        {
            options.ModelValidatorProviders.Add(new NonEmptyGuidValidatorProvider());
            options.ModelMetadataDetailsProviders.Add(new
SystemTextJsonValidationMetadataProvider());
        }).AddJsonOptions(options => ApiJson.Configure(options.JsonSerializerOptions));
    }

    /// <summary>
    /// Adds request context and consistent metadata to a problem-details response.
    /// </summary>
    public static void CustomizeProblem(HttpContext context, ProblemDetails problem)
    {
        problem.Extensions["traceId"] = Activity.Current?.Id ?? context.TraceIdentifier;
        if (problem is ValidationProblemDetails)
        {
            problem.Type = "https://bridgecontrol/errors/validation";
        }
    }
}

```

BoundedLogBodyAttribute.cs

```

using BridgeControl.Services.AgentServices;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using BridgeControl.Security.AgentSecurity;
namespace BridgeControl.Api;

/// <summary>Enforces the log body limit before JSON binding, including chunked requests and
in-memory hosts.</summary>
[AttributeUsage(AttributeTargets.Method)]
public sealed class BoundedLogBodyAttribute : Attribute, IAsyncResourceFilter
{
    /// <summary>
    /// Enforces the log-ingestion request limits before model binding.
    /// </summary>
    public async Task OnResourceExecutionAsync(ResourceExecutingContext context,
ResourceExecutionDelegate next)
    {
        AgentActor actor = AgentActor.From(context.HttpContext.User);
    }
}

```

```

context.HttpContext.RequestServices.GetRequiredService<BridgeControl.Services.AgentServices Agent
RateLimits>().Check("logs", actor.VendorId, actor.AgentId);
    const int limit = 262144;
    HttpRequest request = context.HttpContext.Request;
    void Reject() => context.Result = new ObjectResult(new ProblemDetails { Status =
413, Title = "Log batch is too large." }) { StatusCode = 413 };
    if (request.ContentLength > limit) { Reject(); return; }
    await using MemoryStream buffer = new MemoryStream(); byte[] chunk = new
byte[8192];
    try
    {
        while (true)
        {
            int read = await request.Body.ReadAsync(chunk.AsMemory(0,
Math.Min(chunk.Length, limit + 1 - (int)buffer.Length)), context.HttpContext.RequestAborted);
            if (read == 0) { break; }
            await buffer.WriteAsync(chunk.AsMemory(0, read),
context.HttpContext.RequestAborted);
            if (buffer.Length > limit) { Reject(); return; }
        }
        catch (HttpRequestException e) when (e.StatusCode == 413) { Reject(); return; }
        Stream original = request.Body; buffer.Position = 0; request.Body = buffer;
        try { await next(); } finally { request.Body = original; }
    }
}

```

NonEmptyGuidValidatorProvider.cs

```

using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

namespace BridgeControl.Api;

/// <summary>
/// Rejects empty resource identifiers in action arguments, request properties and collections.
/// </summary>
public sealed class NonEmptyGuidValidatorProvider : IModelValidatorProvider
{
    /// <summary>
    /// Adds validation for GUID values that must not be empty.
    /// </summary>
    public void CreateValidators(ModelValidatorProviderContext context)
    {
        if ((Nullable.GetUnderlyingType(context.ModelMetadata.ModelType) ??
context.ModelMetadata.ModelType) == typeof(Guid))
        {
            context.Results.Add(new ValidatorItem { Validator = new
NonEmptyGuidValidator(), IsReusable = true });
        }
    }

    /// <summary>
    /// Rejects empty GUID values during model validation.
    /// </summary>
    private sealed class NonEmptyGuidValidator : IModelValidator
    {
        /// <summary>
        /// Reports a validation error when a required GUID is empty.
        /// </summary>
    }
}

```

```

        public IEnumerable<ModelValidationResult> Validate(ModelValidationContext context)
            => context.Model is Guid value && value == Guid.Empty
                ? [new ModelValidationResult(null, "An empty identifier is not
valid.")]
                : [];
    }
}

```

PreserveWhitespaceAttribute.cs

```

namespace BridgeControl.Api;

/// <summary>
/// Preserves exact JSON text for passwords, credentials and other whitespace-sensitive input.
/// </summary>
[AttributeUsage(AttributeTargets.Property)]
public sealed class PreserveWhitespaceAttribute : Attribute;

```

StreamMultipartAttribute.cs

```

using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace BridgeControl.Api;

/// <summary>Prevents MVC form providers from buffering/consuming an upload before the bounded
archive reader.</summary>
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public sealed class StreamMultipartAttribute : Attribute, IResourceFilter
{
    /// <summary>
    /// Disables form value providers so multipart uploads can be streamed.
    /// </summary>
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.ValueProviderFactories.RemoveType<FormValueProviderFactory>();
        context.ValueProviderFactories.RemoveType<FormFileValueProviderFactory>();
        context.ValueProviderFactories.RemoveType<JQueryFormValueProviderFactory>();
    }
    /// <summary>
    /// Completes the multipart filter without additional response processing.
    /// </summary>
    public void OnResourceExecuted(ResourceExecutedContext context) { }
}

```

Exceptions

AgentRateLimitException.cs

```

namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Reports agent rate limit failures to the caller.
/// </summary>

```

```
public sealed class AgentRateLimitException : Exception;
```

AgentValidationException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Reports agent validation failures to the caller.
/// </summary>
public sealed class AgentValidationException(string message) : Exception(message);
```

ApiAccessException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Signals a safe authentication, permission or concealed-resource failure.
/// </summary>
public sealed class ApiAccessException(int statusCode) : Exception
{
    public int StatusCode { get; } = statusCode is 401 or 403 or 404 ? statusCode : throw new
ArgumentOutOfRangeException(nameof(statusCode));
}
```

ApiConflictException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Reports an expected business conflict using a public error code and safe explanation.
/// </summary>
public sealed class ApiConflictException(string code, string publicMessage) :
Exception(publicMessage)
{
    public string Code { get; } = code;
}
```

ArtifactValidationException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Only controlled validation messages, never parser errors or private paths.
/// </summary>
public sealed class ArtifactValidationException(string message) : Exception(message);
```

ConfigurationValidationException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Reports configuration validation failures to the caller.
/// </summary>
public sealed class ConfigurationValidationException(string message) : Exception(message);
```

LicenseValidationException.cs

```
namespace BridgeControl.Api.Exceptions;

/// <summary>
/// Reports license validation failures to the caller.
/// </summary>
public sealed class LicenseValidationException(string message) : Exception(message);
```

Controllers

AgentControllers

AgentCommunicationController.cs

```
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.DTOS.AgentDTOS.DeploymentSettings;
using BridgeControl.DTOS.Shared;
using BridgeControl.Repositories.AgentRepositories;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.AgentControllers;

/// <summary>
/// Exposes HTTP endpoints for agent communication.
/// </summary>
[ApiController, Authorize(Policy = AgentAuthentication.Policy), RequestSizeLimit(16384)]
public sealed class AgentCommunicationController(IAgentRepository repository,
Services.AgentServices.AgentAssignmentService assignments) :
ActionController<InstallationResponse>
{
    /// <summary>
    /// Registers agent for the current HTTP request.
    /// </summary>
    [AllowAnonymous, EnableRateLimiting("AgentRegister"), HttpPost("api/v1/agent/register",
Name = "RegisterAgent")]
    [ProducesResponseType<AgentRegistrationResponse>(201)]
    public async Task<ActionResult<AgentRegistrationResponse>> Register(RegisterAgentRequest
request, CancellationToken ct)
        => StatusCode(201, await repository.RegisterAsync(request, ct));

    /// <summary>
    /// Exchanges a valid machine credential for an agent access token.
    /// </summary>
    [AllowAnonymous, EnableRateLimiting("AgentToken"), HttpPost("api/v1/agent/auth/token",
Name = "IssueAgentToken")]
    public async Task<ActionResult<AgentTokenResponse>> Token(AgentTokenRequest request,
CancellationToken ct)
        => Ok(await repository.TokenAsync(request, ct));
```

```

    /// <summary>
    /// Lists agent installations for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/agent/installations", Name = "ListAgentInstallations")]
    public async Task<ActionResult<PagedResponse<AgentInstallationAssignment>>>
List([FromQuery(Name = "")] PaginationQuery page, CancellationToken ct)
        => Ok(await assignments.ListAsync(AgentActor.From(User), page, ct));

    /// <summary>
    /// Registers agent installation for the current HTTP request.
    /// </summary>
    [HttpPost("api/v1/agent/installations", Name = "RegisterAgentInstallation")]
    [ProducesResponseType<InstallationResponse>(201)]
    public Task<ActionResult<InstallationResponse>>
RegisterInstallation(RegisterInstallationRequest request, CancellationToken ct)
        => CreateAsync(request, (value, token) =>
repository.RegisterInstallationAsync(AgentActor.From(User), value, token),
        _ => "/api/v1/agent/installations", ct);

    /// <summary>
    /// Reports agent heartbeat for the current HTTP request.
    /// </summary>
    [HttpPost("api/v1/agent/heartbeat", Name = "ReportAgentHeartbeat")]
    [ProducesResponseType(204)]
    public async Task<IActionResult> Heartbeat(AgentHeartbeatRequest request,
CancellationToken ct)
    {
        await repository.HeartbeatAsync(AgentActor.From(User), request, ct); return
NoContent();
    }
}

```

AgentDeploymentsController.cs

```

using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.DeploymentContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Agents;

/// <summary>
/// Exposes HTTP endpoints for agent deployments.
/// </summary>
[ApiController, Authorize(Policy = AgentAuthentication.Policy),
Route("api/v1/agent/deployments")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class AgentDeploymentsController(IDeploymentRepository repository) : ControllerBase
{
    /// <summary>
    /// Returns agent deployments for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "GetAgentDeployments")]
    public async Task<ActionResult<IReadOnlyList<DeploymentResponse>>> Pending([FromQuery]
PaginationQuery page, CancellationToken ct, string status = "pending")
    {

```

```

        AgentDeploymentPage result = await repository.PendingAsync (AgentActor.From(User),
page, status, ct);
        if (result.HasMore && page.Page < int.MaxValue)
        { Response.Headers.Link =
$"</api/v1/agent/deployments?status=pending&page={page.Page + 1}&pageSize={page.PageSize}>";
rel=\"next\"; }
        return Ok(result.Items);
    }

    /// <summary>
    /// Starts deployment attempt for the current HTTP request.
    /// </summary>
    [HttpPost("{deploymentId}/attempts", Name = "StartDeploymentAttempt")]
    public async Task<ActionResult<StartDeploymentAttemptResponse>> Start(Guid deploymentId,
CancellationTokens ct)
    => Ok(await repository.StartAsync (AgentActor.From(User), deploymentId, ct));

    /// <summary>
    /// Updates deployment attempt for the current HTTP request.
    /// </summary>
    [HttpPatch("{deploymentId}/attempts/{attemptId}", Name = "UpdateDeploymentAttempt"),
RequestSizeLimit (16384)]
    public async Task<ActionResult<DeploymentAttemptResponse>> Update(Guid deploymentId, Guid
attemptId, UpdateDeploymentAttemptRequest request, CancellationTokens ct)
    => Ok(await repository.UpdateAsync (AgentActor.From(User), deploymentId, attemptId,
request, ct));
}

```

AgentInstallationStatesController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.InstallationStateContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Agents;

/// <summary>
/// Exposes HTTP endpoints for agent installation states.
/// </summary>
[ApiController, Authorize(Policy = AgentAuthentication.Policy), Route("api/v1/agent")]
[ResponseCache (NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class AgentInstallationStatesController (IInstallationStateRepository repository) :
ActionController<ReportedStateResponse>
{
    /// <summary>
    /// Returns agent desired state for the current HTTP request.
    /// </summary>
    [HttpGet("installations/{installationId}/desired-state", Name = "GetAgentDesiredState")]
    public async Task<ActionResult<DesiredStateResponse>> Desired(Guid installationId,
CancellationTokens ct)
    {
        AgentDesiredState? result = await
repository.AgentDesiredAsync (AgentActor.From(User), installationId, ct);
        if (result is null) { return Problem(statusCode: 404); }
    }
}

```

```

        // Standard links make authorized downloads discoverable without extending the
        frozen state DTO.
        Response.Headers.Link =
        $"</api/v1/agent/artifacts/{result.InstallerArtifactId}/content>; rel=\"installer\",
        </api/v1/agent/artifacts/{result.ConfigurationArtifactId}/content>; rel=\"configuration\"";
        return Ok(result.State);
    }

    /// <summary>
    /// Reports installation state for the current HTTP request.
    /// </summary>
    [HttpPost("installations/{installationId}/reported-state", Name =
    "ReportInstallationState"), RequestSizeLimit(16384)]
    public Task<ActionResult<ReportedStateResponse>> Report(Guid installationId,
    ReportInstallationStateRequest request, CancellationToken ct)
        => ReadAsync(token => repository.ReportAsync(AgentActor.From(User),
    installationId, request, token), ct);

    /// <summary>
    /// Downloads agent artifact for the current HTTP request.
    /// </summary>
    [HttpGet("artifacts/{artifactId}/content", Name = "DownloadAgentArtifact")]
    public async Task<IAActionResult> Artifact(Guid artifactId, CancellationToken ct)
    {
        AgentArtifactContent download = await
    repository.ArtifactAsync(AgentActor.From(User), artifactId, ct);
        Response.Headers["X-Content-Type-Options"] = "nosniff";
        Response.Headers.ETag = $"\"{download.Artifact.Sha256}\"";
        Response.Headers["Content-Digest"] =
        $"sha-256={Convert.ToBase64String(Convert.FromHexString(download.Artifact.Sha256))}";
        Response.Headers["X-Product-Version-Id"] = download.ProductVersionId.ToString();
        Response.Headers["X-Effective-Configuration-Revision-Id"] =
    download.ConfigurationRevisionId.ToString();
        return File(download.Stream, download.Artifact.ContentType,
    download.Artifact.FileName, enableRangeProcessing: false);
    }
}

```

AgentLogsController.cs

```

using BridgeControl.DTOs.AgentDTOs.OperationalHistoryContracts;
using BridgeControl.DTOs.Vendor.OperationalHistoryContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
namespace BridgeControl.Controllers.Agents;

/// <summary>
/// Exposes HTTP endpoints for agent logs.
/// </summary>
[ApiController, Authorize(Policy = AgentAuthentication.Policy), Route("api/v1/agent/logs")]
public sealed class AgentLogsController(IOperationalHistoryRepository repository) :
    ControllerBase
    {
        /// <summary>
        /// Submits agent logs for the current HTTP request.
        /// </summary>
        [HttpPost(Name = "SubmitAgentLogs"), RequestSizeLimit(262144), Api.BoundedLogBody]
        [ProducesResponseType(204)]
    }

```



```

        public async Task<IActionResult> Submit (SubmitAgentLogsRequest request, CancellationToken
ct)
        {
            if (Request.ContentLength > 262144) { return Problem(statusCode: 413, title: "Log
batch is too large."); }
            await repository.SubmitAsync (AgentActor.From (User), request, ct); return
NoContent ();
        }
    }
}

```

AgentsController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.DTOS.Shared;
using BridgeControl.Repositories.AgentRepositories;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.AgentControllers;

/// <summary>
/// Exposes HTTP endpoints for agents.
/// </summary>
[ApiController, Authorize (Policy = TenantAuthentication.Policy), RequestSizeLimit (16384)]
public sealed class AgentsController (IAgentRepository repository) :
    ActionController<AgentResponse>
{
    /// <summary>
    /// Creates agent enrollment for the current HTTP request.
    /// </summary>
    [HttpPost ("api/v1/agent-enrollments", Name = "CreateAgentEnrollment")]
    [ProducesResponseType<AgentEnrollmentResponse> (201)]
    public async Task<ActionResult<AgentEnrollmentResponse>>
Enroll (CreateAgentEnrollmentRequest request, CancellationToken ct)
        => StatusCode (201, await repository.EnrollAsync (TenantActor.From (User), request,
ct));

    /// <summary>
    /// Lists agents for the current HTTP request.
    /// </summary>
    [HttpGet ("api/v1/agents", Name = "ListAgents")]
    public Task<ActionResult<PagedResponse<AgentResponse>>> List ([FromQuery (Name = "")]
PaginationQuery page, CancellationToken ct)
        => PageAsync (token => repository.PageAsync (TenantActor.From (User), page, token),
ct);

    /// <summary>
    /// Returns agent for the current HTTP request.
    /// </summary>
    [HttpGet ("api/v1/agents/{agentId:guid}", Name = "GetAgent")]
    public Task<ActionResult<AgentResponse>> Get (Guid agentId, CancellationToken ct)
        => ReadOrNotFoundAsync (token => repository.GetAsync (TenantActor.From (User),
agentId, token), ct);

    /// <summary>
    /// Lists agent heartbeats for the current HTTP request.
    /// </summary>

```

```

        [HttpGet("api/v1/agents/{agentId:guid}/heartbeats", Name = "ListAgentHeartbeats")]
        public async Task<ActionResult<PagedResponse<AgentHeartbeatResponse>>> Heartbeats(Guid
agentId, [FromQuery(Name = "")] PaginationQuery page, CancellationToken ct)
            => Ok(await repository.HeartbeatsAsync(TenantActor.From(User), agentId, page,
ct));
    }

```

Controllers/Platform

PlatformAccountController.cs

```

using System.IdentityModel.Tokens.Jwt;
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Platform;
using BridgeControl.Repositories.Platform;
using BridgeControl.Security;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Platform;

/// <summary>
/// Returns the current platform administrator after authentication and platform access checks.
/// </summary>
[ApiController]
[Route("api/v1/platform/account")]
[Tags("Platform account")]
[Authorize(Policy = PlatformAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public class PlatformAccountController(IPlatformAccountRepository accounts) :
    ActionController<PlatformAdminDto>
{
    /// <summary>
    /// Returns current platform administrator for the current HTTP request.
    /// </summary>
    [HttpGet("me", Name = "GetCurrentPlatformAdministrator")]
    [ProducesResponseType<PlatformAdminDto>(StatusCodes.Status200OK)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status403Forbidden)]
    public Task<ActionResult<PlatformAdminDto>> Me(CancellationToken cancellationToken)
    {
        Guid accountId = Guid.Parse(User.FindFirst(JwtRegisteredClaimNames.Sub)!.Value);
        return ReadAsync(token => accounts.GetCurrentAsync(accountId, token),
cancellationToken);
    }
}

```

PlatformAuthController.cs

```

using System.IdentityModel.Tokens.Jwt;
using BridgeControl.DTOs.Platform;
using BridgeControl.Security;
using BridgeControl.Services.Platform;
using Microsoft.AspNetCore.Authorization;

```

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.Platform;

/// <summary>
/// Provides platform login, refresh and logout without exposing stored credential hashes.
/// </summary>
[ApiController]
[Route("api/v1/platform/auth")]
[Tags("Platform authentication")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[Authorize(Policy = PlatformAuthentication.Policy)]
public class PlatformAuthController(PlatformAuthenticationService authentication) :
    ControllerBase
{
    /// <summary>
    /// Authenticates a platform administrator and sets the secure refresh cookie.
    /// </summary>
    [AllowAnonymous]
    [HttpPost("login", Name = "PlatformLogin")]
    [Consumes("application/json")]
    [RequestSizeLimit(4096)]
    [EnableRateLimiting(PlatformAuthentication.RateLimitPolicy)]
    [ProducesResponseType<PlatformAuthSessionResponse>(StatusCodes.Status200OK)]
    [ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status429TooManyRequests)]
    public async Task<ActionResult<PlatformAuthSessionResponse>> Login(PlatformLoginRequest
request, CancellationToken cancellationToken)
    {
        PlatformAuthenticationResult? result = await
authentication.LoginAsync(request.Email, request.Password, cancellationToken);
        return AuthenticationResponse(result);
    }

    /// <summary>
    /// Rotates the platform refresh credential and renews the access token.
    /// </summary>
    [AllowAnonymous]
    [HttpPost("refresh", Name = "PlatformRefresh")]
    [EnableRateLimiting(PlatformAuthentication.RateLimitPolicy)]
    [ProducesResponseType<PlatformAuthSessionResponse>(StatusCodes.Status200OK)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status429TooManyRequests)]
    public async Task<ActionResult<PlatformAuthSessionResponse>> Refresh(CancellationToken
cancellationToken)
    {
        // This non-simple header requires a browser preflight across origins. No
cross-origin access is enabled.
        if (Request.Headers[PlatformRefreshCookie.RequestHeader] != "1")
        {
            return Problem(statusCode: StatusCodes.Status400BadRequest, title: "The
refresh request header is required.");
        }
        PlatformAuthenticationResult? result = await
authentication.RefreshAsync(Request.Cookies[PlatformRefreshCookie.Name], cancellationToken);
        return AuthenticationResponse(result);
    }

    /// <summary>

```

```

    /// Revokes the current platform session and clears its refresh cookie.
    /// </summary>
    [HttpPost("logout", Name = "PlatformLogout")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status403Forbidden)]
    public async Task<IActionResult> Logout(CancellationTokens cancellationTokens)
    {
        Guid accountId = Guid.Parse(User.FindFirst(JwtRegisteredClaimNames.Sub)!.Value);
        Guid sessionId =
        Guid.Parse(User.FindFirst(PlatformAuthentication.SessionIdClaim)!.Value);
        await authentication.LogoutAsync(sessionId, accountId, cancellationTokens);
        PlatformRefreshCookie.Clear(Response);
        return NoContent();
    }

    /// <summary>
    /// Builds the platform session response and writes its refresh cookie.
    /// </summary>
    private ActionResult<PlatformAuthSessionResponse>
    AuthenticationResponse(PlatformAuthenticationResult? result)
    {
        if (result is null)
        {
            PlatformRefreshCookie.Clear(Response);
            return Problem(statusCode: StatusCodes.Status401Unauthorized, title:
            "Authentication failed.");
        }
        PlatformRefreshCookie.Write(Response, result.RefreshCredential,
        result.RefreshExpiresAt);
        return Ok(result.Response);
    }
}

```

PlatformVendorLifecycleController.cs

```

using System.IdentityModel.Tokens.Jwt;
using BridgeControl.Security;
using BridgeControl.Services.Platform;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Platform;

/// <summary>
/// Exposes HTTP endpoints for platform vendor lifecycle.
/// </summary>
[ApiController, Route("api/v1/platform/vendors"), Tags("Platform vendors"), Authorize(Policy =
PlatformAuthentication.Policy)]
[ProducesResponseType(204), ProducesResponseType<ProblemDetails>(401),
ProducesResponseType<ProblemDetails>(403), ProducesResponseType<ProblemDetails>(404),
ProducesResponseType<ProblemDetails>(409)]
public class PlatformVendorLifecycleController(VendorLifecycleService lifecycle) : ControllerBase
{
    /// <summary>
    /// Deactivates vendor for the current HTTP request.
    /// </summary>
    [HttpPost("{vendorId}/deactivate", Name = "DeactivateVendor")]
    public Task<IActionResult> Deactivate(Guid vendorId, CancellationTokens ct) =>
    Change(vendorId, false, ct);
}

```

```

    /// <summary>
    /// Reactivates vendor for the current HTTP request.
    /// </summary>
    [HttpPost("{vendorId}/reactivate", Name = "ReactivateVendor")]
    public Task<IActionResult> Reactivate(Guid vendorId, CancellationToken ct) =>
        Change(vendorId, true, ct);
    /// <summary>
    /// Applies the requested vendor lifecycle transition and returns its HTTP result.
    /// </summary>
    private async Task<IActionResult> Change(Guid id, bool active, CancellationToken ct)
    {
        int status = await lifecycle.SetActiveAsync(id,
            Guid.Parse(User.FindFirst(JwtRegisteredClaimNames.Sub)!.Value), active, ct);
        return status == 204 ? NoContent() : Problem(statusCode: status);
    }
}

```

PlatformVendorsController.cs

```

using System.IdentityModel.Tokens.Jwt;
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Platform;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Repositories.Platform;
using BridgeControl.Security;
using BridgeControl.Services.Platform;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.Platform;

/// <summary>
/// Provides platform-only vendor provisioning, safe vendor reads and initial administrator
/// invitations.
/// </summary>
[ApiController]
[Route("api/v1/platform/vendors")]
[Tags("Platform vendors")]
[Authorize(Policy = PlatformAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
[ProducesResponseType<ProblemDetails>(StatusCodes.Status403Forbidden)]
[ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest)]
public class PlatformVendorsController(IVendorRepository vendors, VendorProvisioningService
    provisioning,
    VendorInvitationService invitations) : ActionController<VendorDto>
{
    /// <summary>
    /// Lists platform vendors for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListPlatformVendors")]
    [ProducesResponseType<PagedResponse<VendorDto>>(StatusCodes.Status200OK)]
    public Task<ActionResult<PagedResponse<VendorDto>>> List([FromQuery] PaginationQuery
        pagination, CancellationToken cancellationToken)
        => PageAsync(token => vendors.PageAsync(pagination, token), cancellationToken);

    /// <summary>
    /// Returns platform vendor for the current HTTP request.

```

```

    /// </summary>
    [HttpGet("{vendorId}", Name = "GetPlatformVendor")]
    [ProducesResponseType<VendorDto>(StatusCodes.Status200OK)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
    public Task<ActionResult<VendorDto>> Get(Guid vendorId, CancellationToken
cancellationToken)
    => ReadOrNotFoundAsync(token => vendors.GetAsync(vendorId, token),
cancellationToken);

    /// <summary>
    /// Creates platform vendor for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "CreatePlatformVendor")]
    [ProducesResponseType<VendorDto>(StatusCodes.Status201Created)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status409Conflict)]
    public Task<ActionResult<VendorDto>> Create(CreateVendorRequest request, CancellationToken
cancellationToken)
    => CreateAsync(request, (input, token) => provisioning.CreateAsync(input,
AdministratorId(), token),
    vendor => $"/api/v1/platform/vendors/{vendor.Id}", cancellationToken);

    /// <summary>
    /// Updates platform vendor for the current HTTP request.
    /// </summary>
    [HttpPatch("{vendorId}", Name = "UpdatePlatformVendor")]
    [ProducesResponseType<VendorDto>(StatusCodes.Status200OK)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status409Conflict)]
    public Task<ActionResult<VendorDto>> Update(Guid vendorId, UpdateVendorRequest request,
CancellationToken cancellationToken)
    => UpdateAsync(request, (input, token) => vendors.UpdateAsync(vendorId, input,
AdministratorId(), token), cancellationToken);

    /// <summary>
    /// Invites initial vendor administrator for the current HTTP request.
    /// </summary>
    [HttpPost("{vendorId}/administrators", Name = "InviteInitialVendorAdministrator")]
    [EnableRateLimiting("VendorInvitation")]
    [ProducesResponseType<UserResponse>(StatusCodes.Status201Created)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status404NotFound)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status409Conflict)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status429TooManyRequests)]
    public async Task<ActionResult<UserResponse>> Invite(Guid vendorId,
CreateVendorAdministratorRequest request, CancellationToken cancellationToken)
    {
        UserResponse? user = await invitations.InviteInitialAsync(vendorId, request,
AdministratorId(), cancellationToken);
        return user is null ? Problem(statusCode: StatusCodes.Status404NotFound) :
Created($"/api/v1/users/{user.Id}", user);
    }

    /// <summary>
    /// Reads the authenticated platform administrator identifier.
    /// </summary>
    private Guid AdministratorId() =>
Guid.Parse(User.FindFirst(JwtRegisteredClaimNames.Sub)!.Value);
}

```

Controllers/Shared

ActionController.cs

```
using BridgeControl.DTOs.Shared;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Shared;

/// <summary>
/// Provides the shared action helpers used by concrete controllers, following BookPub's
/// ActionController naming.
/// </summary>
/// <remarks>
/// Inherited: protected response helpers below. Concrete controllers declare the public HTTP
/// actions,
/// routes, typed request DTOs and authorization rules. Inheriting this class does not create
/// endpoints.
/// </remarks>
public abstract class ActionController<TResponse> : ControllerBase where TResponse : class
{
    /// <summary>
    /// Executes a read operation and returns its result as an HTTP response.
    /// </summary>
    protected async Task<ActionResult<TResponse>> ReadAsync(
        Func<CancellationToken, Task<TResponse>> read, CancellationTok
        cancellationToken)
        => Ok(await read(cancellationToken));

    /// <summary>
    /// Executes a read operation and returns not found when no result exists.
    /// </summary>
    protected async Task<ActionResult<TResponse>> ReadOrNotFoundAsync(
        Func<CancellationToken, Task<TResponse?>> read, CancellationTok
        cancellationToken)
    {
        TResponse? response = await read(cancellationToken);
        return response is null ? Problem(statusCode: StatusCodes.Status404NotFound) :
        Ok(response);
    }

    /// <summary>
    /// Executes a paginated read and adds the corresponding HTTP pagination metadata.
    /// </summary>
    protected async Task<ActionResult<PagedResponse<TResponse>>> PageAsync(
        Func<CancellationToken, Task<PagedResponse<TResponse>>> read, CancellationTok
        cancellationToken)
        => Ok(await read(cancellationToken));

    /// <summary>
    /// Returns a completed creation as 201 with a Location; asynchronous provisioning uses
    /// its own documented response.
    /// </summary>
    protected async Task<ActionResult<TResponse>> CreateAsync<TRequest>(TRequest request,
        Func<TRequest, CancellationToken, Task<TResponse>> create, Func<TResponse, string>
        location,
        CancellationToken cancellationToken) where TRequest : class
    {
        TResponse response = await create(request, cancellationToken);
```

```

        return Created(location(response), response);
    }

    /// <summary>
    /// Returns an updated DTO or a 404; the service enforces ownership, editable fields and
workflow constraints.
    /// </summary>
    protected async Task<ActionResult<TResponse>> UpdateAsync<TRequest>(TRequest request,
        Func<TRequest, CancellationToken, Task<TResponse?>> update,
        CancellationToken cancellationToken) where TRequest : class
    {
        TResponse? response = await update(request, cancellationToken);
        return response is null ? Problem(statusCode: StatusCodes.Status404NotFound) :
Ok(response);
    }

    /// <summary>
    /// Returns 204 after a service-authorized deletion, or 404 when that service reports no
accessible resource.
    /// </summary>
    protected async Task<IActionResult> DeleteAsync(Func<CancellationToken, Task<bool>>
delete,
        CancellationToken cancellationToken)
        => await delete(cancellationToken) ? NoContent() : Problem(statusCode:
StatusCodes.Status404NotFound);
    }

```

Controllers/Vendor

AccessAssignmentsController.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.UserAdministrationRequests;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for access assignments.
/// </summary>
[ApiController, Route("api/v1"), Tags("Roles and access"), Authorize(Policy =
TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403),
ProducesResponseType<ProblemDetails>(404)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(409)]
public class AccessAssignmentsController(IAccessAssignmentRepository assignments) :
ControllerBase
{
    /// <summary>
    /// Lists assignable roles for the current HTTP request.

```



```

    /// </summary>
    [HttpGet("roles", Name = "ListAssignableRoles"),
    ProducesResponseType<PagedResponse<RoleDto>>(200)]
    public Task<PagedResponse<RoleDto>> Roles([FromQuery(Name = "")] PaginationQuery page,
    Cancellation token ct) => assignments.RolesAsync(TenantActor.From(User), page, ct);

    /// <summary>
    /// Lists known permissions for the current HTTP request.
    /// </summary>
    [HttpGet("permissions", Name = "ListKnownPermissions"),
    ProducesResponseType<PagedResponse<PermissionDto>>(200)]
    public Task<PagedResponse<PermissionDto>> Permissions([FromQuery(Name = "")]
    PaginationQuery page, Cancellation token ct) =>
    assignments.PermissionsAsync(TenantActor.From(User), page, ct);

    /// <summary>
    /// Sets user roles for the current HTTP request.
    /// </summary>
    [HttpPut("users/{userId}/roles", Name = "SetUserRoles"), RequestSizeLimit(4096),
    ProducesResponseType(204)]
    public async Task<IActionResult> SetRoles(Guid userId, SetUserRolesRequest request,
    Cancellation token ct)
    {
        await assignments.ReplaceRolesAsync(TenantActor.From(User), userId, request, ct);
        return NoContent();
    }

    /// <summary>
    /// Returns customer access for the current HTTP request.
    /// </summary>
    [HttpGet("users/{userId}/customer-access", Name = "GetCustomerAccess"),
    ProducesResponseType<SetCustomerAccessRequest>(200)]
    public Task<SetCustomerAccessRequest> Customers(Guid userId, Cancellation token ct) =>
    assignments.CustomersAsync(TenantActor.From(User), userId, ct);

    /// <summary>
    /// Sets customer access for the current HTTP request.
    /// </summary>
    [HttpPut("users/{userId}/customer-access", Name = "SetCustomerAccess"),
    RequestSizeLimit(65536), ProducesResponseType(204)]
    public async Task<IActionResult> SetCustomers(Guid userId, SetCustomerAccessRequest
    request, Cancellation token ct)
    {
        await assignments.ReplaceCustomersAsync(TenantActor.From(User), userId, request,
    ct);
        return NoContent();
    }

    /// <summary>
    /// Returns installation access for the current HTTP request.
    /// </summary>
    [HttpGet("users/{userId}/installation-access", Name = "GetInstallationAccess"),
    ProducesResponseType<SetInstallationAccessRequest>(200)]
    public Task<SetInstallationAccessRequest> Installations(Guid userId, Cancellation token ct)
    => assignments.InstallationsAsync(TenantActor.From(User), userId, ct);

    /// <summary>
    /// Sets installation access for the current HTTP request.
    /// </summary>
    [HttpPut("users/{userId}/installation-access", Name = "SetInstallationAccess"),
    RequestSizeLimit(65536), ProducesResponseType(204)]

```

```

        public async Task<IActionResult> SetInstallations(Guid userId,
SetInstallationAccessRequest request, CancellationToken ct)
        {
            await assignments.ReplaceInstallationsAsync(TenantActor.From(User), userId,
request, ct);
            return NoContent();
        }
    }
}

```

AccountActivationController.cs

```

using BridgeControl.DTOS.Vendor;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Activates an invited account using its private, single-use token.
/// </summary>
[ApiController]
[Route("api/v1/auth/activate")]
[Tags("Account activation")]
[AllowAnonymous]
[EnableRateLimiting("AccountActivation")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public class AccountActivationController(AccountActivationService activation) : ControllerBase
{
    /// <summary>
    /// Activates invited account for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "ActivateInvitedAccount")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType<ValidationProblemDetails>(StatusCodes.Status400BadRequest)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status401Unauthorized)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status409Conflict)]
    [ProducesResponseType<ProblemDetails>(StatusCodes.Status429TooManyRequests)]
    public async Task<IActionResult> Activate(ActivateAccountRequest request,
CancellationToken cancellationToken)
    => await activation.ActivateAsync(request, cancellationToken)
        ? NoContent() : Problem(statusCode: StatusCodes.Status401Unauthorized,
title: "The activation token is invalid or unavailable.");
}

```

ConfigurationController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

```

```

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for configuration.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy), Tags("Configuration")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(400), ProducesResponseType<ProblemDetails>(401),
ProducesResponseType<ProblemDetails>(403)]
[ProducesResponseType<ProblemDetails>(404), ProducesResponseType<ProblemDetails>(409)]
public class ConfigurationController(IConfigurationRepository configurations) :
    ActionController<ConfigurationRevisionResponse>
{
    /// <summary>
    /// Returns product configuration for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/products/{productId}/configuration", Name = "GetProductConfiguration"),
    ProducesResponseType<ConfigurationRevisionResponse>(200)]
    public Task<ActionResult<ConfigurationRevisionResponse>> Product(Guid productId,
    Cancellation token ct)
        => ReadOrNotFoundAsync(token =>
    configurations.CurrentAsync(TenantActor.From(User), new(ProductId: productId), token), ct);

    /// <summary>
    /// Sets product configuration for the current HTTP request.
    /// </summary>
    [HttpPut("api/v1/products/{productId}/configuration", Name = "SetProductConfiguration"),
    RequestSizeLimit(ConfigurationValues.MaximumBytes + 4096),
    ProducesResponseType<ConfigurationRevisionResponse>(200)]
    public Task<ActionResult<ConfigurationRevisionResponse>> SetProduct(Guid productId,
    UpdateConfigurationRequest request, Cancellation token ct)
        => ReadAsync(token => configurations.SetAsync(TenantActor.From(User),
    new(ProductId: productId), request, token), ct);

    /// <summary>
    /// Lists product configuration revisions for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/products/{productId}/configuration/revisions", Name =
    "ListProductConfigurationRevisions"),
    ProducesResponseType<PagedResponse<ConfigurationRevisionResponse>>(200)]
    public Task<ActionResult<PagedResponse<ConfigurationRevisionResponse>>>
    ProductHistory(Guid productId, [FromQuery(Name = "")] PaginationQuery page, Cancellation token ct)
        => PageAsync(token => configurations.HistoryAsync(TenantActor.From(User),
    new(ProductId: productId), page, token), ct);

    /// <summary>
    /// Returns customer configuration for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/customers/{customerId}/products/{productId}/configuration", Name =
    "GetCustomerConfiguration"), ProducesResponseType<ConfigurationRevisionResponse>(200)]
    public Task<ActionResult<ConfigurationRevisionResponse>> Customer(Guid customerId, Guid
    productId, Cancellation token ct)
        => ReadOrNotFoundAsync(token =>
    configurations.CurrentAsync(TenantActor.From(User), new(productId, customerId), token), ct);

    /// <summary>
    /// Sets customer configuration for the current HTTP request.
    /// </summary>
    [HttpPut("api/v1/customers/{customerId}/products/{productId}/configuration", Name =
    "SetCustomerConfiguration"), RequestSizeLimit(ConfigurationValues.MaximumBytes + 4096),
    ProducesResponseType<ConfigurationRevisionResponse>(200)]

```

```

        public Task<ActionResult<ConfigurationRevisionResponse>> SetCustomer(Guid customerId, Guid
productId, UpdateConfigurationRequest request, CancellationToken ct)
            => ReadAsync(token => configurations.SetAsync(TenantActor.From(User),
new(productId, customerId), request, token), ct);

        /// <summary>
        /// Lists customer configuration revisions for the current HTTP request.
        /// </summary>
        [HttpGet("api/v1/customers/{customerId}/products/{productId}/configuration/revisions",
Name = "ListCustomerConfigurationRevisions"),
ProducesResponseType<PagedResponse<ConfigurationRevisionResponse>>(200)]
        public Task<ActionResult<PagedResponse<ConfigurationRevisionResponse>>>
CustomerHistory(Guid customerId, Guid productId, [FromQuery(Name = "")] PaginationQuery page,
CancellationToken ct)
            => PageAsync(token => configurations.HistoryAsync(TenantActor.From(User),
new(productId, customerId), page, token), ct);

        /// <summary>
        /// Returns installation configuration for the current HTTP request.
        /// </summary>
        [HttpGet("api/v1/installations/{installationId}/configuration", Name =
"GetInstallationConfiguration"), ProducesResponseType<ConfigurationRevisionResponse>(200)]
        public Task<ActionResult<ConfigurationRevisionResponse>> Installation(Guid installationId,
CancellationToken ct)
            => ReadOrNotFoundAsync(token =>
configurations.CurrentAsync(TenantActor.From(User), new(InstallationId: installationId), token),
ct);

        /// <summary>
        /// Sets installation configuration for the current HTTP request.
        /// </summary>
        [HttpPut("api/v1/installations/{installationId}/configuration", Name =
"SetInstallationConfiguration"), RequestSizeLimit(ConfigurationValues.MaximumBytes + 4096),
ProducesResponseType<ConfigurationRevisionResponse>(200)]
        public Task<ActionResult<ConfigurationRevisionResponse>> SetInstallation(Guid
installationId, UpdateConfigurationRequest request, CancellationToken ct)
            => ReadAsync(token => configurations.SetAsync(TenantActor.From(User),
new(InstallationId: installationId), request, token), ct);

        /// <summary>
        /// Lists installation configuration revisions for the current HTTP request.
        /// </summary>
        [HttpGet("api/v1/installations/{installationId}/configuration/revisions", Name =
"ListInstallationConfigurationRevisions"),
ProducesResponseType<PagedResponse<ConfigurationRevisionResponse>>(200)]
        public Task<ActionResult<PagedResponse<ConfigurationRevisionResponse>>>
InstallationHistory(Guid installationId, [FromQuery(Name = "")] PaginationQuery page,
CancellationToken ct)
            => PageAsync(token => configurations.HistoryAsync(TenantActor.From(User),
new(InstallationId: installationId), page, token), ct);
    }

```

ConfigurationMigrationExecutionController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;

```

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for configuration migration execution.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy),
Route("api/v1/installations/{installationId}/configuration")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class
ConfigurationMigrationExecutionController(IConfigurationMigrationExecutionRepository repository)
: ActionController<ConfigurationMigrationPreviewResponse>
{
    /// <summary>
    /// Previews configuration migration for the current HTTP request.
    /// </summary>
    [HttpPost("migration-preview", Name = "PreviewConfigurationMigration"),
    RequestSizeLimit(16384)]
    public Task<ActionResult<ConfigurationMigrationPreviewResponse>> Preview(Guid
installationId, ConfigurationMigrationPreviewRequest request, CancellationToken ct)
    => ReadAsync(token => repository.PreviewAsync(TenantActor.From(User),
installationId, request, token), ct);

    /// <summary>
    /// Applies configuration migration for the current HTTP request.
    /// </summary>
    [HttpPost("migrate", Name = "ApplyConfigurationMigration"), RequestSizeLimit(16384)]
    public async Task<ActionResult<EffectiveConfigurationResponse>> Apply(Guid installationId,
ApplyConfigurationMigrationRequest request, CancellationToken ct)
    => Ok(await repository.ApplyAsync(TenantActor.From(User), installationId, request,
ct));
}

```

ConfigurationMigrationsController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for configuration migrations.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy),
Route("api/v1/products/{productId}/configuration-migrations")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class ConfigurationMigrationsController(IConfigurationMigrationRepository
migrations) : ActionController<ConfigurationMigrationDto>
{
    /// <summary>
    /// Lists configuration migrations for the current HTTP request.
    /// </summary>

```

```

        [HttpGet(Name = "ListConfigurationMigrations")]
        public Task<ActionResult<PagedResponse<ConfigurationMigrationDto>>> List(Guid productId,
[FromQuery(Name = "")] PaginationQuery page, CancellationTokens ct)
            => PageAsync(token => migrations.ListAsync(TenantActor.From(User), productId,
page, token), ct);

        /// <summary>
        /// Returns configuration migration for the current HTTP request.
        /// </summary>
        [HttpGet("{migrationId}", Name = "GetConfigurationMigration")]
        public Task<ActionResult<ConfigurationMigrationDto>> Get(Guid productId, Guid migrationId,
CancellationTokens ct)
            => ReadOrNotFoundAsync(token => migrations.GetAsync(TenantActor.From(User),
productId, migrationId, token), ct);

        /// <summary>
        /// Creates configuration migration for the current HTTP request.
        /// </summary>
        [HttpPost(Name = "CreateConfigurationMigration"), RequestSizeLimit(1048576)]
        public Task<ActionResult<ConfigurationMigrationDto>> Create(Guid productId,
CreateConfigurationMigrationRequest request, CancellationTokens ct)
            => CreateAsync(request, (input, token) =>
migrations.SaveAsync(TenantActor.From(User), productId, null, input, token), item =>
$"/api/v1/products/{productId}/configuration-migrations/{item.Id}", ct);

        /// <summary>
        /// Replaces configuration migration for the current HTTP request.
        /// </summary>
        [HttpPut("{migrationId}", Name = "ReplaceConfigurationMigration"),
RequestSizeLimit(1048576)]
        public Task<ActionResult<ConfigurationMigrationDto>> Replace(Guid productId, Guid
migrationId, CreateConfigurationMigrationRequest request, CancellationTokens ct)
            => ReadAsync(token => migrations.SaveAsync(TenantActor.From(User), productId,
migrationId, request, token), ct);
    }

```

CustomersController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.UserAdministrationRequests;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for customers.
/// </summary>
[ApiController, Route("api/v1/customers"), Tags("Customers"), Authorize(Policy =
TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403),
ProducesResponseType<ProblemDetails>(404)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(409)]

```

```

public class CustomersController(ICustomerRepository customers) :
    ActionController<CustomerResponse>
{
    /// <summary>
    /// Lists customers for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListCustomers"),
    ProducesResponseType<PagedResponse<CustomerResponse>>(200)]
    public Task<ActionResult<PagedResponse<CustomerResponse>>> List([FromQuery(Name = "")]
    PaginationQuery page, [FromQuery] CustomerQuery filter, CancellationToken ct)
        => PageAsync(token => customers.PageAsync(TenantActor.From(User), page, filter,
    token), ct);

    /// <summary>
    /// Returns customer for the current HTTP request.
    /// </summary>
    [HttpGet("{customerId}", Name = "GetCustomer"),
    ProducesResponseType<CustomerResponse>(200)]
    public Task<ActionResult<CustomerResponse>> Get(Guid customerId, CancellationToken ct)
        => ReadOrNotFoundAsync(token => customers.GetAsync(TenantActor.From(User),
    customerId, token), ct);

    /// <summary>
    /// Creates customer for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "CreateCustomer"), RequestSizeLimit(4096),
    ProducesResponseType<CustomerResponse>(201)]
    public Task<ActionResult<CustomerResponse>> Create(CreateCustomerRequest request,
    CancellationToken ct)
        => CreateAsync(request, (input, token) =>
    customers.CreateAsync(TenantActor.From(User), input, token), customer =>
    $"api/v1/customers/{customer.Id}", ct);

    /// <summary>
    /// Updates customer for the current HTTP request.
    /// </summary>
    [HttpPatch("{customerId}", Name = "UpdateCustomer"), RequestSizeLimit(4096),
    ProducesResponseType<CustomerResponse>(200)]
    public Task<ActionResult<CustomerResponse>> Update(Guid customerId, UpdateCustomerRequest
    request, CancellationToken ct)
        => UpdateAsync(request, (input, token) =>
    customers.UpdateAsync(TenantActor.From(User), customerId, input, token), ct);

    /// <summary>
    /// Deactivates customer for the current HTTP request.
    /// </summary>
    [HttpPost("{customerId}/deactivate", Name = "DeactivateCustomer"),
    ProducesResponseType(204)]
    public async Task<IActionResult> Deactivate(Guid customerId, CancellationToken ct)
        => await customers.SetActiveAsync(TenantActor.From(User), customerId, false, ct) ?
    NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Reactivates customer for the current HTTP request.
    /// </summary>
    [HttpPost("{customerId}/reactivate", Name = "ReactivateCustomer"),
    ProducesResponseType(204)]
    public async Task<IActionResult> Reactivate(Guid customerId, CancellationToken ct)
        => await customers.SetActiveAsync(TenantActor.From(User), customerId, true, ct) ?
    NoContent() : Problem(statusCode: 404);
}

```

DeploymentsController.cs

```
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.DeploymentContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for deployments.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy), Route("api/v1/deployments")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class DeploymentsController(IDeploymentRepository repository) :
    ActionController<DeploymentResponse>
{
    /// <summary>
    /// Searches deployments for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "SearchDeployments")]
    public Task<ActionResult<PagedResponse<DeploymentResponse>>> Search([FromQuery]
        PaginationQuery page, Guid? installationId, Guid? customerId, string? status, CancellationToken
        ct)
        => PageAsync(token => repository.SearchAsync(TenantActor.From(User), page,
            installationId, customerId, status, token), ct);

    /// <summary>
    /// Returns installation deployments for the current HTTP request.
    /// </summary>
    [HttpGet("/api/v1/installations/{installationId}/deployments", Name =
        "GetInstallationDeployments")]
    public Task<ActionResult<PagedResponse<DeploymentResponse>>> History(Guid installationId,
        [FromQuery] PaginationQuery page, string? status, CancellationToken ct)
        => PageAsync(token => repository.SearchAsync(TenantActor.From(User), page,
            installationId, null, status, token), ct);

    /// <summary>
    /// Returns deployment for the current HTTP request.
    /// </summary>
    [HttpGet("{deploymentId}", Name = "GetDeployment")]
    public async Task<ActionResult<DeploymentDetailResponse>> Detail(Guid deploymentId,
        CancellationToken ct)
    {
        DeploymentDetailResponse? result = await
            repository.DetailAsync(TenantActor.From(User), deploymentId, ct);
        return result is null ? Problem(statusCode: 404) : Ok(result);
    }

    /// <summary>
    /// Cancels deployment for the current HTTP request.
    /// </summary>
    [HttpPost("{deploymentId}/cancel", Name = "CancelDeployment")]
}
```



```

        public Task<ActionResult<DeploymentResponse>> Cancel(Guid deploymentId, CancellationToken
ct)
        => ReadAsync(token => repository.CancelAsync(TenantActor.From(User), deploymentId,
token), ct);
    }

```

DeploymentSettingsController.cs

```

using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for deployment settings.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy), RequestSizeLimit(8192)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class DeploymentSettingsController(DeploymentSettingsService settings) :
ControllerBase
{
    /// <summary>
    /// Returns product deployment settings for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/products/{productId}/deployment-settings", Name =
"GetProductDeploymentSettings")]
    public Task<DeploymentSettingsResponse> Product(Guid productId, CancellationToken ct) =>
settings.GetAsync(TenantActor.From(User), productId, null, ct);

    /// <summary>
    /// Sets product deployment settings for the current HTTP request.
    /// </summary>
    [HttpPut("api/v1/products/{productId}/deployment-settings", Name =
"SetProductDeploymentSettings")]
    public Task<DeploymentSettingsResponse> SetProduct(Guid productId, DeploymentSettings
request, CancellationToken ct) => settings.PutAsync(TenantActor.From(User), productId, null,
request, false, ct);

    /// <summary>
    /// Returns version deployment settings for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/products/{productId}/versions/{versionId}/deployment-settings", Name =
"GetVersionDeploymentSettings")]
    public Task<DeploymentSettingsResponse> Version(Guid productId, Guid versionId,
CancellationToken ct) => settings.GetAsync(TenantActor.From(User), productId, versionId, ct);

    /// <summary>
    /// Sets version deployment settings for the current HTTP request.
    /// </summary>
    [HttpPut("api/v1/products/{productId}/versions/{versionId}/deployment-settings", Name =
"SetVersionDeploymentSettings")]
    public Task<DeploymentSettingsResponse> SetVersion(Guid productId, Guid versionId,
DeploymentSettings request, CancellationToken ct) => settings.PutAsync(TenantActor.From(User),
productId, versionId, request, false, ct);

    /// <summary>

```

```

        /// Initializes published deployment settings for the current HTTP request.
        /// </summary>

[HttpPost("api/v1/products/{productId}/versions/{versionId}/deployment-settings/initialize", Name
= "InitializePublishedDeploymentSettings")]
    public Task<DeploymentSettingsResponse> Initialize(Guid productId, Guid versionId,
DeploymentSettings request, CancellationToken ct) => settings.PutAsync(TenantActor.From(User),
productId, versionId, request, true, ct);
}

```

EffectiveConfigurationsController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for effective configurations.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy),
Route("api/v1/installations/{installationId}/effective-configuration")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class EffectiveConfigurationsController(IEffectiveConfigurationRepository
repository) : ActionController<EffectiveConfigurationResponse>
{
    /// <summary>
    /// Returns effective configuration for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "GetEffectiveConfiguration")]
    public Task<ActionResult<EffectiveConfigurationResponse>> Current(Guid installationId,
CancellationToken ct)
        => ReadOrNotFoundAsync(token => repository.CurrentAsync(TenantActor.From(User),
installationId, token), ct);

    /// <summary>
    /// Lists effective configuration revisions for the current HTTP request.
    /// </summary>
    [HttpGet("revisions", Name = "ListEffectiveConfigurationRevisions")]
    public Task<ActionResult<PagedResponse<EffectiveConfigurationResponse>>> History(Guid
installationId, [FromQuery(Name = "")] PaginationQuery page, CancellationToken ct)
        => PageAsync(token => repository.HistoryAsync(TenantActor.From(User),
installationId, page, token), ct);

    /// <summary>
    /// Returns effective configuration revision for the current HTTP request.
    /// </summary>
    [HttpGet("revisions/{revisionId}", Name = "GetEffectiveConfigurationRevision")]
    public Task<ActionResult<EffectiveConfigurationResponse>> Revision(Guid installationId,
Guid revisionId, CancellationToken ct)
        => ReadOrNotFoundAsync(token => repository.GetAsync(TenantActor.From(User),
installationId, revisionId, token), ct);
}

```

InstallationsController.cs

```
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.AgentDTOs.AgentContracts;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.DTOs.Vendor.LicenseContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for installations.
/// </summary>
[ApiController, Route("api/v1/installations"), Authorize(Policy = TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class InstallationsController(IInstallationRepository repository) :
    ActionController<InstallationResponse>
{
    /// <summary>
    /// Assigns product to agent for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "AssignProductToAgent"), RequestSizeLimit(4096),
    ProducesResponseType<InstallationResponse>(201)]
    public Task<ActionResult<InstallationResponse>> Create(AssignProductRequest request,
        CancellationToken ct)
        => CreateAsync(request, (value, token) =>
        repository.CreateAsync(TenantActor.From(User), value, token), value =>
        $"/api/v1/installations/{value.Id}", ct);

    /// <summary>
    /// Lists installations for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListInstallations")]
    public Task<ActionResult<PagedResponse<InstallationResponse>>> List([FromQuery(Name = "")]
        PaginationQuery page, [FromQuery] InstallationQuery filter, CancellationToken ct)
        => PageAsync(token => repository.PageAsync(TenantActor.From(User), page, filter,
        token), ct);

    /// <summary>
    /// Returns installation for the current HTTP request.
    /// </summary>
    [HttpGet("{installationId}", Name = "GetInstallation")]
    public Task<ActionResult<InstallationResponse>> Get(Guid installationId, CancellationToken
    ct)
        => ReadOrNotFoundAsync(token => repository.GetAsync(TenantActor.From(User),
        installationId, token), ct);

    /// <summary>
    /// Updates installation for the current HTTP request.
    /// </summary>
    [HttpPatch("{installationId}", Name = "UpdateInstallation"), RequestSizeLimit(4096)]
    public Task<ActionResult<InstallationResponse>> Update(Guid installationId,
        UpdateInstallationRequest request, CancellationToken ct)
        => UpdateAsync(request, (input, token) =>
        repository.UpdateAsync(TenantActor.From(User), installationId, input, token), ct);
```

```

    /// <summary>
    /// Deactivates installation for the current HTTP request.
    /// </summary>
    [HttpPost("{installationId}/deactivate", Name = "DeactivateInstallation"),
    ProducesResponseType(204)]
    public async Task<IActionResult> Deactivate(Guid installationId, CancellationToken ct)
        => await repository.SetActiveAsync(TenantActor.From(User), installationId, false,
        ct) ? NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Reactivates installation for the current HTTP request.
    /// </summary>
    [HttpPost("{installationId}/reactivate", Name = "ReactivateInstallation"),
    ProducesResponseType(204)]
    public async Task<IActionResult> Reactivate(Guid installationId, CancellationToken ct)
        => await repository.SetActiveAsync(TenantActor.From(User), installationId, true,
        ct) ? NoContent() : Problem(statusCode: 404);
}

```

InstallationStatesController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.InstallationStateContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for installation states.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy),
Route("api/v1/installations/{installationId}")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class InstallationStatesController(IInstallationStateRepository repository) :
    ActionController<DesiredStateResponse>
{
    /// <summary>
    /// Returns desired state for the current HTTP request.
    /// </summary>
    [HttpGet("desired-state", Name = "GetDesiredState")]
    public Task<ActionResult<DesiredStateResponse>> Desired(Guid installationId,
        CancellationToken ct)
        => ReadOrNotFoundAsync(token => repository.DesiredAsync(TenantActor.From(User),
        installationId, token), ct);

    /// <summary>
    /// Sets desired state for the current HTTP request.
    /// </summary>
    [HttpPut("desired-state", Name = "SetDesiredState"), RequestSizeLimit(16384)]
    public Task<ActionResult<DesiredStateResponse>> Set(Guid installationId,
        SetDesiredStateRequest request, CancellationToken ct)
        => ReadAsync(token => repository.SetDesiredAsync(TenantActor.From(User),
        installationId, request, token), ct);
}

```

```

    /// <summary>
    /// Returns reported state for the current HTTP request.
    /// </summary>
    [HttpGet("reported-state", Name = "GetReportedState")]
    public async Task<ActionResult<ReportedStateResponse>> Reported(Guid installationId,
CancellationTokens ct)
    {
        ReportedStateResponse? result = await
repository.ReportedAsync(TenantActor.From(User), installationId, ct);
        return result is null ? Problem(statusCode: 404) : Ok(result);
    }
}

```

LicensesController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.LicenseContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for licenses.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy), RequestSizeLimit(8192)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class LicensesController(ILicenseRepository repository) :
    ActionController<LicenseResponse>
{
    /// <summary>
    /// Lists licenses for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/licenses", Name = "ListLicenses")]
    public Task<ActionResult<PagedResponse<LicenseResponse>>> List([FromQuery(Name = "")]
PaginationQuery page, [FromQuery] InstallationQuery filter, CancellationTokens ct)
    => PageAsync(token => repository.PageAsync(TenantActor.From(User), page, filter,
token), ct);

    /// <summary>
    /// Returns license for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/licenses/{licenseId}", Name = "GetLicense")]
    public Task<ActionResult<LicenseResponse>> Get(Guid licenseId, CancellationTokens ct)
    => ReadOrNotFoundAsync(token => repository.GetAsync(TenantActor.From(User),
licenseId, token), ct);

    /// <summary>
    /// Creates license for the current HTTP request.
    /// </summary>
    [HttpPost("api/v1/licenses", Name = "CreateLicense"),
ProducesResponseType<LicenseResponse>(201)]
    public Task<ActionResult<LicenseResponse>> Create(CreateLicenseRequest request,
CancellationTokens ct)

```

```

        => CreateAsync(request, (input, token) =>
repository.CreateAsync(TenantActor.From(User), input, token), license =>
$/api/v1/licenses/{license.Id}", ct);

    /// <summary>
    /// Updates license for the current HTTP request.
    /// </summary>
    [HttpPatch("api/v1/licenses/{licenseId}", Name = "UpdateLicense")]
    public Task<ActionResult<LicenseResponse>> Update(Guid licenseId, UpdateLicenseRequest
request, CancellationToken ct)
        => UpdateAsync(request, (input, token) =>
repository.UpdateAsync(TenantActor.From(User), licenseId, input, token), ct);

    /// <summary>
    /// Deactivates license for the current HTTP request.
    /// </summary>
    [HttpPost("api/v1/licenses/{licenseId}/deactivate", Name = "DeactivateLicense"),
ProducesResponseType(204)]
    public async Task<IAActionResult> Deactivate(Guid licenseId, CancellationToken ct)
        => await repository.SetActiveAsync(TenantActor.From(User), licenseId, false, ct) ?
NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Reactivates license for the current HTTP request.
    /// </summary>
    [HttpPost("api/v1/licenses/{licenseId}/reactivate", Name = "ReactivateLicense"),
ProducesResponseType(204)]
    public async Task<IAActionResult> Reactivate(Guid licenseId, CancellationToken ct)
        => await repository.SetActiveAsync(TenantActor.From(User), licenseId, true, ct) ?
NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Assigns installation license for the current HTTP request.
    /// </summary>
    [HttpPut("api/v1/installations/{installationId}/license", Name =
"AssignInstallationLicense"), ProducesResponseType(204)]
    public async Task<IAActionResult> Assign(Guid installationId, AssignLicenseRequest request,
CancellationToken ct)
        => await repository.AssignAsync(TenantActor.From(User), installationId,
request.LicenseId, ct) ? NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Removes installation license for the current HTTP request.
    /// </summary>
    [HttpDelete("api/v1/installations/{installationId}/license", Name =
"RemoveInstallationLicense"), ProducesResponseType(204)]
    public async Task<IAActionResult> Remove(Guid installationId, CancellationToken ct)
        => await repository.AssignAsync(TenantActor.From(User), installationId, null, ct)
? NoContent() : Problem(statusCode: 404);
}

```

OperationalHistoryController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.OperationalHistoryContracts;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;

```

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for operational history.
/// </summary>
[ApiController, Authorize(Policy = TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
public sealed class OperationalHistoryController(IOperationalHistoryRepository repository) :
    ActionController<LogResponse>
{
    /// <summary>
    /// Searches logs for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/logs", Name = "SearchLogs")]
    public Task<ActionResult<PagedResponse<LogResponse>>> Search([FromQuery] PaginationQuery
page, [FromQuery] LogSearchQuery search, CancellationToken ct)
        => PageAsync(token => repository.LogsAsync(TenantActor.From(User), page, search,
null, null, token), ct);

    /// <summary>
    /// Returns installation logs for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/installations/{installationId}/logs", Name = "GetInstallationLogs")]
    public Task<ActionResult<PagedResponse<LogResponse>>> Installation(Guid installationId,
[FromQuery] PaginationQuery page, [FromQuery] LogSearchQuery search, CancellationToken ct)
        => PageAsync(token => repository.LogsAsync(TenantActor.From(User), page, search,
installationId, null, token), ct);

    /// <summary>
    /// Returns agent logs for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/agents/{agentId}/logs", Name = "GetAgentLogs")]
    public Task<ActionResult<PagedResponse<LogResponse>>> Agent(Guid agentId, [FromQuery]
PaginationQuery page, [FromQuery] LogSearchQuery search, CancellationToken ct)
        => PageAsync(token => repository.LogsAsync(TenantActor.From(User), page, search,
null, agentId, token), ct);

    /// <summary>
    /// Returns monitoring overview for the current HTTP request.
    /// </summary>
    [HttpGet("api/v1/monitoring/overview", Name = "GetMonitoringOverview")]
    public async Task<ActionResult<MonitoringOverviewResponse>> Overview([FromQuery]
PaginationQuery page, CancellationToken ct)
        => Ok(await repository.OverviewAsync(TenantActor.From(User), page, ct));
}

```

ProductArtifactsController.cs

```

using BridgeControl.Api;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Options;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.WebUtilities;

```

```

using Microsoft.Net.Http.Headers;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for product artifacts.
/// </summary>
[ApiController, StreamMultipart, Route("api/v1/products/{productId}/versions/{versionId}"),
Tags("Product artifacts"), Authorize(Policy = TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(400), ProducesResponseType<ProblemDetails>(401),
ProducesResponseType<ProblemDetails>(403)]
[ProducesResponseType<ProblemDetails>(404), ProducesResponseType<ProblemDetails>(409)]
public class ProductArtifactsController(ProductArtifactRepository artifacts,
VendorArtifactArchive archive) : ControllerBase
{
    /// <summary>
    /// Uploads installer for the current HTTP request.
    /// </summary>
    [HttpPost("installer", Name = "UploadInstaller"), Consumes("multipart/form-data"),
RequestSizeLimit(ArtifactArchiveOptions.MaximumInstallerBytes + 65536),
ProducesResponseType<FileArtifactDto>(200)]
    public Task<ActionResult<FileArtifactDto>> PutInstaller(Guid productId, Guid versionId,
CancellationToken ct) => Upload(productId, versionId, "Installer", ct);

    /// <summary>
    /// Uploads schema for the current HTTP request.
    /// </summary>
    [HttpPut("schema", Name = "UploadSchema"), Consumes("multipart/form-data"),
RequestSizeLimit(1024 * 1024 + 65536), ProducesResponseType<FileArtifactDto>(200)]
    public Task<ActionResult<FileArtifactDto>> PutSchema(Guid productId, Guid versionId,
CancellationToken ct) => Upload(productId, versionId, "Schema", ct);

    /// <summary>
    /// Returns installer for the current HTTP request.
    /// </summary>
    [HttpGet("installer", Name = "GetInstaller"), ProducesResponseType<FileArtifactDto>(200)]
    public Task<ActionResult<FileArtifactDto>> GetInstaller(Guid productId, Guid versionId,
CancellationToken ct) => Metadata(productId, versionId, "Installer", ct);

    /// <summary>
    /// Returns schema for the current HTTP request.
    /// </summary>
    [HttpGet("schema", Name = "GetSchema"), ProducesResponseType<FileArtifactDto>(200)]
    public Task<ActionResult<FileArtifactDto>> GetSchema(Guid productId, Guid versionId,
CancellationToken ct) => Metadata(productId, versionId, "Schema", ct);

    /// <summary>
    /// Downloads installer for the current HTTP request.
    /// </summary>
    [HttpGet("installer/content", Name = "DownloadInstaller"), Produces("application/x-msi"),
ProducesResponseType(200)]
    public Task<IActionResult> InstallerContent(Guid productId, Guid versionId,
CancellationToken ct) => Content(productId, versionId, "Installer", ct);

    /// <summary>
    /// Downloads schema for the current HTTP request.
    /// </summary>
    [HttpGet("schema/content", Name = "DownloadSchema"), Produces("application/schema+json"),
ProducesResponseType(200)]
    public Task<IActionResult> SchemaContent(Guid productId, Guid versionId, CancellationToken
ct) => Content(productId, versionId, "Schema", ct);

```



```

    /// <summary>
    /// Deletes installer for the current HTTP request.
    /// </summary>
    [HttpDelete("installer", Name = "DeleteInstaller"), ProducesResponseType(204)]
    public async Task<IActionResult> DeleteInstaller(Guid productId, Guid versionId,
CancellationTokens ct)
    {
        await artifacts.DeleteAsync(TenantActor.From(User), productId, versionId,
"Installer", ct);
        return NoContent();
    }

    /// <summary>
    /// Deletes schema for the current HTTP request.
    /// </summary>
    [HttpDelete("schema", Name = "DeleteSchema"), ProducesResponseType(204)]
    public async Task<IActionResult> DeleteSchema(Guid productId, Guid versionId,
CancellationTokens ct)
    {
        await artifacts.DeleteAsync(TenantActor.From(User), productId, versionId,
"Schema", ct);
        return NoContent();
    }

    /// <summary>
    /// Returns metadata for the selected version artifact.
    /// </summary>
    private async Task<ActionResult<FileArtifactDto>> Metadata(Guid productId, Guid versionId,
string kind, CancellationTokens ct)
    {
        FileArtifactDto? artifact = await artifacts.GetAsync(TenantActor.From(User),
productId, versionId, kind, ct);
        return artifact is null ? Problem(statusCode: StatusCodes.Status404NotFound,
title: "Resource not found.") : Ok(artifact);
    }

    /// <summary>
    /// Streams verified content for the selected version artifact.
    /// </summary>
    private async Task<IActionResult> Content(Guid productId, Guid versionId, string kind,
CancellationTokens ct)
    {
        (Stream Stream, FileArtifactDto Metadata) download = await
artifacts.ContentAsync(TenantActor.From(User), productId, versionId, kind, ct);
        Response.Headers["X-Content-Type-Options"] = "nosniff";
        return File(download.Stream, download.Metadata.ContentType,
download.Metadata.FileName, enableRangeProcessing: false);
    }

    /// <summary>
    /// Reads the multipart upload and stores the validated version artifact.
    /// </summary>
    private async Task<ActionResult<FileArtifactDto>> Upload(Guid productId, Guid versionId,
string kind, CancellationTokens ct)
    {
        CancellationTokens requestAborted = ct;
        using CancellationTokensSource timeout =
CancellationTokensSource.CreateLinkedTokensSource(ct);
        timeout.CancelAfter(archive.UploadTimeout);
        ct = timeout.Token;
        try

```

```

        {
            if (!MediaTypeHeaderValue.TryParse(Request.ContentType, out
MediaTyHeaderValue? type) || type.MediaType != "multipart/form-data")
            { throw new ArtifactValidationException("Use multipart/form-data with one
file field named file."); }
            string? boundary = HeaderUtilities.RemoveQuotes(type.Boundary).Value;
            if (string.IsNullOrEmpty(boundary) || boundary.Length > 128) { throw
new ArtifactValidationException("A valid multipart boundary is required."); }
            // No IFormFile binding: untrusted bytes stream directly into the vendor's
private stage.
            MultipartReader reader = new(boundary, Request.Body) { HeadersCountLimit =
8, HeadersLengthLimit = 8192 };
            MultipartSection? section = await reader.ReadNextSectionAsync(ct);
            if (section is null ||
!ContentDispositionHeaderValue.TryParse(section.ContentDisposition, out
ContentDispositionHeaderValue? disposition)
                || disposition.DispositionType != "form-data" ||
HeaderUtilities.RemoveQuotes(disposition.Name).Value != "file"
                || (!disposition.FileName.HasValue &&
!disposition.FileNameStar.HasValue))
            { throw new ArtifactValidationException("Supply exactly one file field
named file."); }
            string name =
HeaderUtilities.RemoveQuotes(disposition.FileNameStar.HasValue ? disposition.FileNameStar :
disposition.FileName).Value!;
            return await artifacts.PutAsync(TenantActor.From(User), productId,
versionId, kind, name, section.Body, async token =>
            {
                if (await reader.ReadNextSectionAsync(token) is not null) { throw
new ArtifactValidationException("Only one file and no additional form fields are accepted."); }
                }, ct);
            }
            catch (InvalidDataException) { throw new ArtifactValidationException("Malformed or
incomplete multipart upload."); }
            catch (IOException) when (!ct.IsCancellationRequested) { throw new
ArtifactValidationException("The upload could not be read completely. Retry the upload."); }
            catch (OperationCanceledException) when (!requestAborted.IsCancellationRequested)
            { throw new ArtifactValidationException("The upload exceeded its time limit."); }
        }
    }
}

```

ProductsController.cs

```

using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for products.
/// </summary>
[ApiController, Route("api/v1/products"), Tags("Products"), Authorize(Policy =
TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]

```

```

[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403),
ProducesResponseType<ProblemDetails>(404)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(409)]
public class ProductsController(ProductRepository products) : ActionController<ProductDto>
{
    /// <summary>
    /// Lists products for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListProducts"), ProducesResponseType<PagedResponse<ProductDto>>(200)]
    public Task<ActionResult<PagedResponse<ProductDto>>> List([FromQuery(Name = "")]
PaginationQuery page, CancellationToken ct)
        => PageAsync(token => products.PageAsync(TenantActor.From(User), page, token),
ct);

    /// <summary>
    /// Returns product for the current HTTP request.
    /// </summary>
    [HttpGet("{productId}", Name = "GetProduct"), ProducesResponseType<ProductDto>(200)]
    public Task<ActionResult<ProductDto>> Get(Guid productId, CancellationToken ct)
        => ReadOrNotFoundAsync(token => products.GetAsync(TenantActor.From(User),
productId, token), ct);

    /// <summary>
    /// Creates product for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "CreateProduct"), RequestSizeLimit(16384),
ProducesResponseType<ProductDto>(201)]
    public Task<ActionResult<ProductDto>> Create(CreateProductRequest request,
CancellationToken ct)
        => CreateAsync(request, (input, token) =>
products.CreateAsync(TenantActor.From(User), input, token), product =>
$/api/v1/products/{product.Id}", ct);

    /// <summary>
    /// Updates product for the current HTTP request.
    /// </summary>
    [HttpPatch("{productId}", Name = "UpdateProduct"), RequestSizeLimit(16384),
ProducesResponseType<ProductDto>(200)]
    public Task<ActionResult<ProductDto>> Update(Guid productId, UpdateProductRequest request,
CancellationToken ct)
        => UpdateAsync(request, (input, token) =>
products.UpdateAsync(TenantActor.From(User), productId, input, token), ct);

    /// <summary>
    /// Deactivates product for the current HTTP request.
    /// </summary>
    [HttpPost("{productId}/deactivate", Name = "DeactivateProduct"),
ProducesResponseType(204)]
    public async Task<IActionResult> Deactivate(Guid productId, CancellationToken ct)
        => await products.SetActiveAsync(TenantActor.From(User), productId, false, ct) ?
NoContent() : Problem(statusCode: 404);

    /// <summary>
    /// Reactivates product for the current HTTP request.
    /// </summary>
    [HttpPost("{productId}/reactivate", Name = "ReactivateProduct"),
ProducesResponseType(204)]
    public async Task<IActionResult> Reactivate(Guid productId, CancellationToken ct)
        => await products.SetActiveAsync(TenantActor.From(User), productId, true, ct) ?
NoContent() : Problem(statusCode: 404);
}

```

ProductVersionsController.cs

```
using BridgeControl.Controllers.Shared;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for product versions.
/// </summary>
[ApiController, Route("api/v1/products/{productId}/versions"), Tags("Product versions"),
Authorize(Policy = TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403),
ProducesResponseType<ProblemDetails>(404)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(409)]
public class ProductVersionsController(ProductRepository products, IVersionPublishingRepository
publishing) : ActionController<ProductVersionDto>
{
    /// <summary>
    /// Publishes product version for the current HTTP request.
    /// </summary>
    [HttpPost("{versionId}/publish", Name = "PublishProductVersion"),
ProducesResponseType<ProductVersionDto>(200)]
    public Task<ActionResult<ProductVersionDto>> Publish(Guid productId, Guid versionId,
CancellationToken ct)
    => ReadAsync(token => publishing.PublishAsync(TenantActor.From(User), productId,
versionId, token), ct);

    /// <summary>
    /// Lists product versions for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListProductVersions"),
ProducesResponseType<PagedResponse<ProductVersionDto>>(200)]
    public Task<ActionResult<PagedResponse<ProductVersionDto>>> List(Guid productId,
[FromQuery(Name = "")] PaginationQuery page, CancellationToken ct)
    => PageAsync(token => products.VersionsAsync(TenantActor.From(User), productId,
page, token), ct);

    /// <summary>
    /// Returns product version for the current HTTP request.
    /// </summary>
    [HttpGet("{versionId}", Name = "GetProductVersion"),
ProducesResponseType<ProductVersionDto>(200)]
    public Task<ActionResult<ProductVersionDto>> Get(Guid productId, Guid versionId,
CancellationToken ct)
    => ReadOrNotFoundAsync(token => products.VersionAsync(TenantActor.From(User),
productId, versionId, token), ct);

    /// <summary>
    /// Creates product version for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "CreateProductVersion"), RequestSizeLimit(4096),
ProducesResponseType<ProductVersionDto>(201)]
```

```

        public Task<ActionResult<ProductVersionDto>> Create(Guid productId,
CreateProductVersionRequest request, CancellationToken ct)
            => CreateAsync(request, (input, token) =>
products.CreateVersionAsync(TenantActor.From(User), productId, input, token), version =>
$"/api/v1/products/{productId}/versions/{version.Id}", ct);

        /// <summary>
        /// Updates product version for the current HTTP request.
        /// </summary>
        [HttpPatch("{versionId}", Name = "UpdateProductVersion"), RequestSizeLimit(4096),
ProducesResponseType<ProductVersionDto>(200)]
        public Task<ActionResult<ProductVersionDto>> Update(Guid productId, Guid versionId,
UpdateProductVersionRequest request, CancellationToken ct)
            => UpdateAsync(request, (input, token) =>
products.UpdateVersionAsync(TenantActor.From(User), productId, versionId, input, token), ct);
    }

```

TenantAuthController.cs

```

using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.TenantAuthenticationDtos;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for tenant auth.
/// </summary>
[ApiController, Route("api/v1/auth"), Tags("Vendor and customer authentication")]
[Authorize(Policy = TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(429)]
public class TenantAuthController(TenantAuthenticationService authentication,
TenantPasswordService passwords) : ControllerBase
{
    /// <summary>
    /// Authenticates a tenant user and sets the secure refresh cookie.
    /// </summary>
    [AllowAnonymous, HttpPost("login", Name = "TenantLogin"),
EnableRateLimiting("TenantLogin"), RequestSizeLimit(4096)]
    [ProducesResponseType<AuthSessionResponse>(200)]
    public async Task<ActionResult<AuthSessionResponse>> Login(LoginRequest request,
CancellationToken ct)
        => SessionResponse(await authentication.LoginAsync(request, ct));

    /// <summary>
    /// Rotates the tenant refresh credential and renews the access token.
    /// </summary>
    [AllowAnonymous, HttpPost("refresh", Name = "TenantRefresh"),
EnableRateLimiting("TenantRefresh")]
    [ProducesResponseType<AuthSessionResponse>(200)]
    public async Task<ActionResult<AuthSessionResponse>> Refresh(CancellationToken ct)
    {
        if (Request.Headers[TenantRefreshCookie.RequestHeader] != "1") { return
Problem(statusCode: 400, title: "The refresh request header is required."); }
    }

```

```

        return SessionResponse(await
authentication.RefreshAsync(Request.Cookies[TenantRefreshCookie.Name], ct));
    }

    /// <summary>
    /// Revokes the current tenant session and clears its refresh cookie.
    /// </summary>
    [HttpPost("logout", Name = "TenantLogout"), ProducesResponseType(204)]
    public async Task<IActionResult> Logout(CancellationToken ct)
    {
        bool result = await
authentication.LogoutAsync(TenantAuthentication.VendorId(User),
TenantAuthentication.UserId(User), TenantAuthentication.SessionId(User), ct);
        TenantRefreshCookie.Clear(Response);
        return result ? NoContent() : Problem(statusCode: 401, title: "Authentication
failed.");
    }

    /// <summary>
    /// Requests a password-reset invitation without revealing account existence.
    /// </summary>
    [AllowAnonymous, HttpPost("password/forgot", Name = "PasswordForgot"),
EnableRateLimiting("PasswordForgot"), RequestSizeLimit(4096)]
    [ProducesResponseType(204)]
    public async Task<IActionResult> Forgot(ForgotPasswordRequest request, CancellationToken
ct)
    {
        await passwords.ForgotAsync(request, ct); return NoContent(); }

    /// <summary>
    /// Resets a password using a valid single-use reset credential.
    /// </summary>
    [AllowAnonymous, HttpPost("password/reset", Name = "PasswordReset"),
EnableRateLimiting("PasswordReset"), RequestSizeLimit(4096)]
    [ProducesResponseType(204)]
    public async Task<IActionResult> Reset(ResetPasswordRequest request, CancellationToken ct)
    {
        bool success = await passwords.ResetAsync(request, ct);
        if (success) { TenantRefreshCookie.Clear(Response); }
        return success ? NoContent() : Problem(statusCode: 401, title: "Authentication
failed.");
    }

    /// <summary>
    /// Changes the authenticated user's password after verifying the current password.
    /// </summary>
    [HttpPost("password/change", Name = "PasswordChange"),
EnableRateLimiting("PasswordChange"), RequestSizeLimit(4096)]
    [ProducesResponseType(204)]
    public async Task<IActionResult> Change(ChangePasswordRequest request, CancellationToken
ct)
    {
        bool success = await passwords.ChangeAsync(TenantAuthentication.VendorId(User),
TenantAuthentication.UserId(User), TenantAuthentication.SessionId(User), request, ct);
        if (success) { TenantRefreshCookie.Clear(Response); }
        return success ? NoContent() : Problem(statusCode: 401, title: "Authentication
failed.");
    }

    /// <summary>
    /// Builds the tenant session response and writes its refresh cookie.
    /// </summary>

```

```

        private ActionResult<AuthSessionResponse> SessionResponse(TenantAuthenticationResult?
result)
        {
            if (result is null) { TenantRefreshCookie.Clear(Response); return
Problem(statusCode: 401, title: "Authentication failed."); }
            TenantRefreshCookie.Write(Response, result.RefreshCredential,
result.RefreshExpiresAt);
            return Ok(result.Response);
        }
    }
}

```

UserLifecycleController.cs

```

using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for user lifecycle.
/// </summary>
[ApiController, Route("api/v1/users"), Tags("Users"), Authorize(Policy =
TenantAuthentication.Policy)]
[ProducesResponseType(204), ProducesResponseType<ProblemDetails>(401),
ProducesResponseType<ProblemDetails>(403), ProducesResponseType<ProblemDetails>(404)]
public class UserLifecycleController(UserLifecycleService lifecycle) : ControllerBase
{
    /// <summary>
    /// Deactivates user for the current HTTP request.
    /// </summary>
    [HttpPost("{userId}/deactivate", Name = "DeactivateUser")]
    public Task<IActionResult> Deactivate(Guid userId, CancellationToken ct) => Change(userId,
false, ct);

    /// <summary>
    /// Reactivates user for the current HTTP request.
    /// </summary>
    [HttpPost("{userId}/reactivate", Name = "ReactivateUser")]
    public Task<IActionResult> Reactivate(Guid userId, CancellationToken ct) => Change(userId,
true, ct);

    /// <summary>
    /// Applies the requested user lifecycle transition and returns its HTTP result.
    /// </summary>
    private async Task<IActionResult> Change(Guid id, bool active, CancellationToken ct)
    {
        int status = await lifecycle.SetActiveAsync(TenantAuthentication.VendorId(User),
TenantAuthentication.UserId(User), TenantAuthentication.SessionId(User), id, active, ct);
        return status == 204 ? NoContent() : Problem(statusCode: status);
    }
}

```

UsersController.cs

```

using BridgeControl.Controllers.Shared;

```

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.UserAdministrationRequests;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Security;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;

namespace BridgeControl.Controllers.Vendor;

/// <summary>
/// Exposes HTTP endpoints for users.
/// </summary>
[ApiController, Route("api/v1/users"), Tags("Users"), Authorize(Policy =
TenantAuthentication.Policy)]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[ProducesResponseType<ProblemDetails>(401), ProducesResponseType<ProblemDetails>(403),
ProducesResponseType<ProblemDetails>(404)]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(409)]
public class UsersController(IUserAdministrationRepository users) :
ActionController<UserResponse>
{
    /// <summary>
    /// Lists users for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListUsers"), ProducesResponseType<PagedResponse<UserResponse>>(200)]
    public Task<ActionResult<PagedResponse<UserResponse>>> List([FromQuery(Name = "")]
PaginationQuery page, [FromQuery] UserQuery filter, CancellationToken ct)
=> PageAsync(token => users.PageAsync(TenantActor.From(User), page, filter,
token), ct);

    /// <summary>
    /// Returns user for the current HTTP request.
    /// </summary>
    [HttpGet("{userId}", Name = "GetUser"), ProducesResponseType<UserResponse>(200)]
    public Task<ActionResult<UserResponse>> Get(Guid userId, CancellationToken ct)
=> ReadOrNotFoundAsync(token => users.GetAsync(TenantActor.From(User), userId,
token), ct);

    /// <summary>
    /// Invites user for the current HTTP request.
    /// </summary>
    [HttpPost(Name = "InviteUser"), EnableRateLimiting("VendorInvitation"),
RequestSizeLimit(4096)]
    [ProducesResponseType<UserResponse>(201), ProducesResponseType<ProblemDetails>(429)]
    public Task<ActionResult<UserResponse>> Invite(CreateUserRequest request,
CancellationToken ct)
=> CreateAsync(request, (input, token) =>
users.InviteAsync(TenantActor.From(User), input, token), user => $"/api/v1/users/{user.Id}", ct);

    /// <summary>
    /// Updates user for the current HTTP request.
    /// </summary>
    [HttpPatch("{userId}", Name = "UpdateUser"), EnableRateLimiting("VendorInvitation"),
RequestSizeLimit(4096), ProducesResponseType<UserResponse>(200),
ProducesResponseType<ProblemDetails>(429)]
    public Task<ActionResult<UserResponse>> Update(Guid userId, UpdateUserRequest request,
CancellationToken ct)
=> UpdateAsync(request, (input, token) =>
users.UpdateAsync(TenantActor.From(User), userId, input, token), ct);

```



```
}
```

VendorDiscoveryController.cs

```
using BridgeControl.Data;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Models.Platform;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.RateLimiting;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Controllers.Vendor;

/// <summary>Exposes only active vendor names for the pre-login vendor selector.</summary>
[ApiController, AllowAnonymous, Route("api/v1/auth/vendors"), Tags("Vendor and customer authentication")]
[ResponseCache(NoStore = true, Location = ResponseCacheLocation.None)]
[EnableRateLimiting("VendorDiscovery")]
[ProducesResponseType<ValidationProblemDetails>(400), ProducesResponseType<ProblemDetails>(429)]
public sealed class VendorDiscoveryController(PlatformDbContext database) : ControllerBase
{
    /// <summary>
    /// Lists login vendors for the current HTTP request.
    /// </summary>
    [HttpGet(Name = "ListLoginVendors"),
    ProducesResponseType<PagedResponse<LoginVendorResponse>>(200)]
    public async Task<ActionResult<PagedResponse<LoginVendorResponse>>> List([FromQuery]
    PaginationQuery page, CancellationToken ct)
    {
        IQueryable<SoftwareVendor> vendors =
        database.SoftwareVendors.AsNoTracking().Where(vendor => vendor.Status == "Active");
        int count = await vendors.CountAsync(ct);
        List<LoginVendorResponse> names = await vendors.OrderBy(vendor =>
        vendor.Name).ThenBy(vendor => vendor.Id)
        .Skip(page.Offset).Take(page.PageSize).Select(vendor => new
        LoginVendorResponse(vendor.Name)).ToListAsync(ct);
        return Ok(new PagedResponse<LoginVendorResponse> { Items = names, Page =
        page.Page, PageSize = page.PageSize, TotalCount = count });
    }
}
```

Data/Configurations/AgentConfigurations

AgentConfiguration.cs

```
using BridgeControl.Models.AgentModels;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Data.Configurations.AgentConfigurations;
```

```

/// <summary>
/// Configures database mappings and constraints for agent entities.
/// </summary>
public static class AgentConfiguration
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Agent"/>.
    /// </summary>
    public static void Configure(ModelBuilder model)
    {
        model.Entity<Agent>(b =>
        {
            b.HasAlternateKey(x => new { x.Id, x.CustomerId });
            b.HasOne(x => x.Customer).WithMany().HasForeignKey(x =>
x.CustomerId).OnDelete(DeleteBehavior.NoAction);
        });
        model.Entity<AgentEnrollment>(b =>
        {
            b.HasOne(x => x.Customer).WithMany().HasForeignKey(x =>
x.CustomerId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<Agent>().WithMany().HasForeignKey(x => new { x.AgentId,
x.CustomerId })
                .HasPrincipalKey(x => new { x.Id, x.CustomerId
}).OnDelete(DeleteBehavior.NoAction);
        });
        model.Entity<AgentCredential>().HasOne(x => x.Agent).WithMany().HasForeignKey(x =>
x.AgentId).OnDelete(DeleteBehavior.NoAction);
        model.Entity<AgentHeartbeat>(b =>
        {
            b.HasOne<Agent>().WithMany().HasForeignKey(x =>
x.AgentId).OnDelete(DeleteBehavior.NoAction);
            b.HasIndex(x => new { x.AgentId, x.ReportedAt }).IsUnique();
        });
        model.Entity<Installation>(b =>
        {
            b.HasOne<Agent>().WithMany().HasForeignKey(x => new { x.AgentId,
x.CustomerId })
                .HasPrincipalKey(x => new { x.Id, x.CustomerId
}).OnDelete(DeleteBehavior.NoAction);
            b.Property(x => x.RegistrationName).HasMaxLength(200);
            b.HasIndex(x => new { x.CustomerId, x.ProductId, x.RegistrationName
}).IsUnique().HasFilter("[AgentId] IS NOT NULL");
            b.Property(x => x.Status).HasMaxLength(32).HasDefaultValue("Active");
            // Existing unassigned installations receive the migration timestamp, not a
            fabricated historical date.
            b.Property(x => x.RegisteredAt).HasDefaultValueSql("SYSDATETIMEOFFSET()");
            b.Property(x => x.InstalledVersion).HasMaxLength(100);
        });
    }
}

```

Data/Configurations/Platform

PlatformAdminConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Defines database storage for PlatformAdmin records.
/// </summary>
public class PlatformAdminConfiguration : IEntityTypeConfiguration<PlatformAdmin>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="PlatformAdmin"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<PlatformAdmin> builder)
    {
        builder.ToTable("PlatformAdmins");

        builder.Property(entity => entity.Email)
            .HasMaxLength(254)
            .IsRequired();

        builder.Property(entity => entity.PasswordHash)
            .HasMaxLength(512)
            .IsRequired();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.LastLoginAt)
            .HasPrecision(7);

        builder.HasMany(entity => entity.Sessions)
            .WithOne()
            .HasForeignKey(session => session.AccountId)
            .IsRequired()
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

PlatformAuditLogConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Defines database storage for PlatformAuditLog records.
/// </summary>
public class PlatformAuditLogConfiguration : IEntityTypeConfiguration<PlatformAuditLog>
{
    /// <summary>

```

```

        /// Configures the table, properties, relationships, and constraints for <see
    cref="PlatformAuditLog"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<PlatformAuditLog> builder)
    {
        builder.ToTable("PlatformAuditLogs");

        builder.Property(entity => entity.Action)
            .HasMaxLength(100)
            .IsRequired();

        builder.Property(entity => entity.EntityType)
            .HasMaxLength(100)
            .IsRequired();

        builder.Property(entity => entity.EntityId)
            .HasMaxLength(200)
            .IsRequired();

        builder.Property(entity => entity.Details)
            .HasColumnType("nvarchar(max)")
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        // Preserve dependent records; deleting relationships must be an explicit
operation.
        builder.HasOne(entity => entity.PlatformAdmin)
            .WithMany(entity => entity.AuditLogs)
            .HasForeignKey(entity => entity.PlatformAdminId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.Vendor)
            .WithMany(entity => entity.AuditLogs)
            .HasForeignKey(entity => entity.VendorId)
            // Do not automatically clear tracked optional links when the parent is
deleted.
            .OnDelete(DeleteBehavior.ClientNoAction);
    }
}

```

SoftwareVendorConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Defines vendor storage and the unique, case-insensitive name used during login.
/// </summary>
public class SoftwareVendorConfiguration : IEntityTypeConfiguration<SoftwareVendor>
{
    // Match string.Trim's Unicode whitespace characters, including tabs and non-breaking
spaces.
    private const string TrimmedNameSql =
        "TRIM(NCHAR(9) + NCHAR(10) + NCHAR(11) + NCHAR(12) + NCHAR(13) + NCHAR(32) + " +

```

```

        "NCHAR(133) + NCHAR(160) + NCHAR(5760) + NCHAR(8192) + NCHAR(8193) + NCHAR(8194) +
" +
        "NCHAR(8195) + NCHAR(8196) + NCHAR(8197) + NCHAR(8198) + NCHAR(8199) + NCHAR(8200)
+ " +
        "NCHAR(8201) + NCHAR(8202) + NCHAR(8232) + NCHAR(8233) + NCHAR(8239) + NCHAR(8287)
+ " +
        "NCHAR(12288) FROM [Name])";

    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="SoftwareVendor"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<SoftwareVendor> builder)
    {
        builder.Property(entity => entity.RowVersion).IsRowVersion();
        builder.ToTable("SoftwareVendors", table =>
            table.HasCheckConstraint("CK_SoftwareVendors_Name_TrimmedNotEmpty",
                "DATALENGTH([Name]) > 0 AND DATALENGTH([Name]) = DATALENGTH(" +
TrimmedNameSql + ")"));

        builder.Property(entity => entity.Name)
            .HasMaxLength(200)
            .UseCollation("Latin1_General_100_CI_AS_SC")
            // The converter also trims parameters in EF name-equality queries.
            .HasConversion(value => value.Trim(), value => value)
            .IsRequired();

        builder.HasIndex(entity => entity.Name)
            .IsUnique();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.DeactivatedAt)
            .HasPrecision(7);
    }
}

```

VendorDatabaseConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Defines database storage for VendorDatabase records.
/// </summary>
public class VendorDatabaseConfiguration : IEntityTypeConfiguration<VendorDatabase>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="VendorDatabase"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<VendorDatabase> builder)
    {
    }
}

```

```

{
    builder.ToTable("VendorDatabases");

    builder.Property(entity => entity.DatabaseName)
        .HasMaxLength(128)
        .IsRequired();

    builder.Property(entity => entity.ConnectionSecretRef)
        .HasMaxLength(512)
        .IsRequired();

    builder.Property(entity => entity.Status)
        .HasMaxLength(32)
        .IsRequired();

    builder.Property(entity => entity.CreatedAt)
        .HasPrecision(7);

    // Preserve dependent records; deleting relationships must be an explicit
    operation.

    builder.HasOne(entity => entity.Vendor)
        .WithOne(entity => entity.Database)
        .HasForeignKey<VendorDatabase>(entity => entity.VendorId)
        .OnDelete(DeleteBehavior.NoAction);
}
}

```

VendorFileArchiveConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Defines database storage for VendorFileArchive records.
/// </summary>
public class VendorFileArchiveConfiguration : IEntityTypeConfiguration<VendorFileArchive>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="VendorFileArchive"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<VendorFileArchive> builder)
    {
        builder.ToTable("VendorFileArchives");

        builder.Property(entity => entity.StorageIdentifier)
            .HasMaxLength(1024)
            .IsRequired();

        builder.Property(entity => entity.AccessSecretRef)
            .HasMaxLength(512)
            .IsRequired();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();
    }
}

```

```

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        // Preserve dependent records; deleting relationships must be an explicit
operation.
        builder.HasOne(entity => entity.Vendor)
            .WithOne(entity => entity.FileArchive)
            .HasForeignKey<VendorFileArchive>(entity => entity.VendorId)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

VendorMaintenanceOperationConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Configures database mappings and constraints for vendor maintenance operation entities.
/// </summary>
public sealed class VendorMaintenanceOperationConfiguration :
    IEntityTypeConfiguration<VendorMaintenanceOperation>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="VendorMaintenanceOperation"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<VendorMaintenanceOperation> builder)
    {
        builder.HasKey(x => x.Id);
        builder.Property(x => x.RowVersion).IsRowVersion();
        builder.Property(x => x.Kind).HasMaxLength(16);
        builder.Property(x => x.Status).HasMaxLength(16);
        builder.Property(x => x.Stage).HasMaxLength(150);
        builder.Property(x => x.SourceSchema).HasMaxLength(150);
        builder.Property(x => x.CurrentSchema).HasMaxLength(150);
        builder.Property(x => x.TargetSchema).HasMaxLength(150);
        builder.Property(x => x.FailureCode).HasMaxLength(64);
        builder.Property(x => x.BundleSha256).HasMaxLength(64);
        builder.Property(x => x.OperatorAccount).HasMaxLength(256);
        builder.Property(x => x.OperatorHost).HasMaxLength(256);
        builder.HasIndex(x => x.VendorId).IsUnique().HasFilter("[IsActive] = 1");
        builder.HasIndex(x => new { x.Kind, x.Status, x.CompletedAt });
        builder.HasOne<SoftwareVendor>().WithMany().HasForeignKey(x =>
x.VendorId).OnDelete(DeleteBehavior.NoAction);
        builder.HasOne<PlatformAdmin>().WithMany().HasForeignKey(x =>
x.PlatformAdminId).OnDelete(DeleteBehavior.NoAction);
    }
}

```

VendorProvisioningConfiguration.cs

```

using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;

```

```

using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Platform;

/// <summary>
/// Configures database mappings and constraints for vendor provisioning entities.
/// </summary>
public class VendorProvisioningConfiguration : IEntityTypeConfiguration<VendorProvisioning>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="VendorProvisioning"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<VendorProvisioning> builder)
    {
        builder.HasKey(entity => entity.VendorId);
        builder.Property(entity => entity.SchemaVersion).HasMaxLength(150);
        builder.Property(entity => entity.FailureCode).HasMaxLength(64);
        builder.HasOne(entity => entity.Vendor).WithOne(entity => entity.Provisioning)
            .HasForeignKey<VendorProvisioning>(entity =>
entity.VendorId).OnDelete(DeleteBehavior.NoAction);
    }
}

```

Data/Configurations/Shared

AuthenticationSessionConfiguration.cs

```

using BridgeControl.Models.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Shared;

/// <summary>
/// Defines session storage and concurrency rules reused by both database contexts.
/// </summary>
public class AuthenticationSessionConfiguration : IEntityTypeConfiguration<AuthenticationSession>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="AuthenticationSession"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<AuthenticationSession> builder)
    {
        builder.ToTable("AuthenticationSessions", table =>
            table.HasCheckConstraint("CK_AuthenticationSessions_Expiry",
                "[ExpiresAt] > [CreatedAt]"));

        builder.HasKey(entity => entity.Id);

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.ExpiresAt)
            .HasPrecision(7);
    }
}

```



```

        builder.Property(entity => entity.LastRefreshedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.RevokedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.RowVersion)
            .IsRowVersion();

        // Each account configuration defines its own AccountId foreign key.
    }
}

```

RefreshTokenConfiguration.cs

```

using BridgeControl.Models.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Shared;

/// <summary>
/// Defines refresh-token hashes, session ownership and concurrency rules for both databases.
/// </summary>
public class RefreshTokenConfiguration : IEntityTypeConfiguration<RefreshToken>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="RefreshToken"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<RefreshToken> builder)
    {
        builder.ToTable("RefreshTokens", table =>
        {
            table.HasCheckConstraint("CK_RefreshTokens_TokenHash",
                "DATALENGTH([TokenHash]) = 32");
            table.HasCheckConstraint("CK_RefreshTokens_Expiry",
                "[ExpiresAt] > [CreatedAt]");
        });

        builder.HasKey(entity => entity.Id);

        builder.Property(entity => entity.TokenHash)
            .HasMaxLength(32)
            .IsRequired();

        builder.HasIndex(entity => entity.TokenHash)
            .IsUnique();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.ExpiresAt)
            .HasPrecision(7);

        builder.Property(entity => entity.UsedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.RowVersion)

```

```

        .IsRowVersion();

        builder.HasOne(entity => entity.AuthenticationSession)
            .WithMany(entity => entity.RefreshTokens)
            .HasForeignKey(entity => entity.AuthenticationSessionId)
            .IsRequired()
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

Data/Configurations/Vendor

AccountActivationConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for account activation entities.
/// </summary>
public class AccountActivationConfiguration : IEntityTypeConfiguration<AccountActivation>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="AccountActivation"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<AccountActivation> builder)
    {
        builder.ToTable("AccountActivations", table =>
        {
            table.HasCheckConstraint("CK_AccountActivations_Expiry", "[ExpiresAt] > [CreatedAt]");
            builder.Property(entity =>
            entity.TokenHash).HasMaxLength(32).IsFixedLength().IsRequired();
            builder.Property(entity => entity.ProtectedToken).HasMaxLength(2048);
            builder.Property(entity => entity.RowVersion).IsRowVersion();
            builder.HasIndex(entity => entity.UserId).IsUnique().HasFilter("[UsedAt] IS NULL
            AND [RevokedAt] IS NULL");
            builder.HasOne(entity => entity.User).WithMany().HasForeignKey(entity =>
            entity.UserId).OnDelete(DeleteBehavior.NoAction);
        }
    }
}

```

ConfigurationMigrationConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for ConfigurationMigration records.

```

```

/// </summary>
public class ConfigurationMigrationConfiguration :
IEntityTypeConfiguration<ConfigurationMigration>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="ConfigurationMigration"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<ConfigurationMigration> builder)
    {
        builder.ToTable("ConfigurationMigrations");

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        // Preserve dependent records; deleting relationships must be an explicit
operation.
        builder.HasOne(entity => entity.FromProductVersion)
            .WithMany(entity => entity.OutgoingMigrations)
            .HasForeignKey(entity => entity.FromProductId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.ToProductVersion)
            .WithMany(entity => entity.IncomingMigrations)
            .HasForeignKey(entity => entity.ToProductId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.CreatedByUser)
            .WithMany(entity => entity.ConfigurationMigrations)
            .HasForeignKey(entity => entity.CreatedById)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

ConfigurationProfileConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for ConfigurationProfile records.
/// </summary>
public class ConfigurationProfileConfiguration : IEntityTypeConfiguration<ConfigurationProfile>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="ConfigurationProfile"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<ConfigurationProfile> builder)
    {
        builder.Property(x => x.RowVersion).IsRowVersion();
        builder.ToTable("ConfigurationProfiles", table =>
            table.HasCheckConstraint("CK_ConfigurationProfiles_Scope",
                "([ProductId] IS NOT NULL AND [InstallationId] IS NULL) OR " +
                "([InstallationId] IS NOT NULL AND [ProductId] IS NULL AND
[CustomerId] IS NULL)");
    }
}

```

```

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        // One profile per scope; changes are stored as revisions within that profile.
        builder.HasIndex(entity => entity.ProductId)
            .IsUnique()
            .HasFilter("[ProductId] IS NOT NULL AND [CustomerId] IS NULL AND
[InstallationId] IS NULL");

        builder.HasIndex(entity => new { entity.CustomerId, entity.ProductId })
            .IsUnique()
            .HasFilter("[CustomerId] IS NOT NULL AND [ProductId] IS NOT NULL AND
[InstallationId] IS NULL");

        builder.HasIndex(entity => entity.InstallationId)
            .IsUnique()
            .HasFilter("[InstallationId] IS NOT NULL");

        // Preserve dependent records; deleting relationships must be an explicit
operation.
        builder.HasOne(entity => entity.Product)
            .WithMany(entity => entity.ConfigurationProfiles)
            .HasForeignKey(entity => entity.ProductId)
            // Do not automatically clear tracked optional links when the parent is
deleted.

            .OnDelete(DeleteBehavior.ClientNoAction);

        builder.HasOne(entity => entity.Customer)
            .WithMany(entity => entity.ConfigurationProfiles)
            .HasForeignKey(entity => entity.CustomerId)
            // Do not automatically clear tracked optional links when the parent is
deleted.

            .OnDelete(DeleteBehavior.ClientNoAction);

        builder.HasOne(entity => entity.Installation)
            .WithMany(entity => entity.ConfigurationProfiles)
            .HasForeignKey(entity => entity.InstallationId)
            // Do not automatically clear tracked optional links when the parent is
deleted.

            .OnDelete(DeleteBehavior.ClientNoAction);
    }
}

```

ConfigurationRevisionConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for ConfigurationRevision records.
/// </summary>
public class ConfigurationRevisionConfiguration : IEntityTypeConfiguration<ConfigurationRevision>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="ConfigurationRevision"/>.
    /// </summary>

```

```

        public void Configure(EntityTypeBuilder<ConfigurationRevision> builder)
        {
            builder.ToTable("ConfigurationRevisions", t => {
                t.HasTrigger("TR_ConfigurationRevisions_Immutable"); t.UseSqlOutputClause(false); });
            builder.Property(x => x.ValidationStatus).HasMaxLength(32).IsRequired();
            builder.HasOne<FileArtifact>().WithMany().HasForeignKey(x =>
                x.SchemaArtifactId).onDelete(DeleteBehavior.NoAction);
            builder.HasIndex(x => new { x.ConfigurationProfileId, x.ProductVersionId,
                x.RevisionNumber });

            builder.Property(entity => entity.CreatedAt)
                .HasPrecision(7);

            builder.HasIndex(entity => new { entity.ConfigurationProfileId,
                entity.RevisionNumber })
                .IsUnique();

            // Preserve dependent records; deleting relationships must be an explicit
            operation.
            builder.HasOne(entity => entity.ConfigurationProfile)
                .WithMany(entity => entity.Revisions)
                .HasForeignKey(entity => entity.ConfigurationProfileId)
                .onDelete(DeleteBehavior.NoAction);

            builder.HasOne(entity => entity.ProductVersion)
                .WithMany(entity => entity.ConfigurationRevisions)
                .HasForeignKey(entity => entity.ProductVersionId)
                .onDelete(DeleteBehavior.NoAction);

            builder.HasOne(entity => entity.FileArtifact)
                .WithMany(entity => entity.ConfigurationRevisions)
                .HasForeignKey(entity => entity.FileArtifactId)
                .onDelete(DeleteBehavior.NoAction);

            builder.HasOne(entity => entity.CreatedByUser)
                .WithMany(entity => entity.ConfigurationRevisions)
                .HasForeignKey(entity => entity.CreatedById)
                .onDelete(DeleteBehavior.NoAction);
        }
    }
}

```

CustomerConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for Customer records.
/// </summary>
public class CustomerConfiguration : IEntityTypeConfiguration<Customer>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Customer"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<Customer> builder)
    {

```

```

        builder.ToTable("Customers");
        builder.Property(entity => entity.RowVersion).IsRowVersion();

        builder.Property(entity => entity.Name)
            .HasMaxLength(200)
            .IsRequired();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.DeactivatedAt)
            .HasPrecision(7);
    }
}

```

CustomerUserInstallationAccessConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for customer user installation access entities.
/// </summary>
public class CustomerUserInstallationAccessConfiguration :
    IEntityTypeConfiguration<CustomerUserInstallationAccess>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="CustomerUserInstallationAccess"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<CustomerUserInstallationAccess> builder)
    {
        builder.HasIndex(entity => new { entity.UserId, entity.InstallationId
    }).IsUnique();
        builder.HasOne(entity => entity.User).WithMany().HasForeignKey(entity =>
entity.UserId).OnDelete(DeleteBehavior.NoAction);
        builder.HasOne(entity => entity.GrantedByUser).WithMany().HasForeignKey(entity =>
entity.GrantedByUserId).OnDelete(DeleteBehavior.NoAction);
        builder.HasOne(entity => entity.Installation).WithMany()
            .HasForeignKey(entity => new { entity.InstallationId, entity.CustomerId })
            .HasPrincipalKey(entity => new { entity.Id, entity.CustomerId
    }).OnDelete(DeleteBehavior.NoAction);
        // The migration adds the matching UserId/CustomerId FK against a unique index.
        // An EF alternate key would incorrectly make UserAccount.CustomerId required for
        vendor users.
    }
}

```

EffectiveConfigurationRevisionConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for EffectiveConfigurationRevision records.
/// </summary>
public class EffectiveConfigurationRevisionConfiguration :
    IEntityTypeConfiguration<EffectiveConfigurationRevision>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="EffectiveConfigurationRevision"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<EffectiveConfigurationRevision> builder)
    {
        builder.ToTable("EffectiveConfigurationRevisions", t => {
            t.HasTrigger("TR_EffectiveConfigurationRevisions_Immutable"); t.UseSqlOutputClause(false); });

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.HasIndex(entity => new { entity.InstallationId, entity.RevisionNumber })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
operation.
        builder.HasOne(entity => entity.Installation)
            .WithMany(entity => entity.EffectiveConfigurationRevisions)
            .HasForeignKey(entity => new { entity.InstallationId, entity.ProductId })
            .HasPrincipalKey(entity => new { entity.Id, entity.ProductId })
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.ProductVersion)
            .WithMany(entity => entity.EffectiveConfigurationRevisions)
            .HasForeignKey(entity => new { entity.ProductVersionId, entity.ProductId })
            .HasPrincipalKey(entity => new { entity.Id, entity.ProductId })
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.FileArtifact)
            .WithMany(entity => entity.EffectiveConfigurationRevisions)
            .HasForeignKey(entity => entity.FileArtifactId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.CreatedByUser)
            .WithMany(entity => entity.EffectiveConfigurationRevisions)
            .HasForeignKey(entity => entity.CreatedByUserId)
            // Do not automatically clear tracked optional links when the parent is
deleted.
            .OnDelete(DeleteBehavior.ClientNoAction);
    }
}

```

EffectiveConfigurationSourceRevisionConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

```

```

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for EffectiveConfigurationSourceRevision records.
/// </summary>
public class EffectiveConfigurationSourceRevisionConfiguration :
IEntityTypeConfiguration<EffectiveConfigurationSourceRevision>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="EffectiveConfigurationSourceRevision"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<EffectiveConfigurationSourceRevision> builder)
    {
        builder.ToTable("EffectiveConfigurationSourceRevisions", t => {
t.HasTrigger("TR_EffectiveConfigurationSources_Immutable"); t.UseSqlOutputClause(false); });

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.EffectiveConfigurationRevision)
            .WithMany(entity => entity.SourceRevisions)
            .HasForeignKey(entity => entity.EffectiveConfigurationRevisionId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.ConfigurationRevision)
            .WithMany(entity => entity.EffectiveConfigurationSources)
            .HasForeignKey(entity => entity.ConfigurationRevisionId)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

EffectiveConfigurationWorkConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for effective configuration work entities.
/// </summary>
public static class EffectiveConfigurationWorkConfiguration
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="EffectiveConfigurationWork"/>.
    /// </summary>
    public static void Configure(ModelBuilder model)
    {
        model.Entity<EffectiveConfigurationSignal>(b =>
        {
            b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.ProductVersionId).OnDelete(DeleteBehavior.NoAction);
            b.HasIndex(x => new { x.ProcessedAt, x.CreatedAt });
            b.ToTable("EffectiveConfigurationSignals", t =>
t.HasCheckConstraint("CK_EffectiveSignal_Target",

```



```

        "([InstallationId] IS NULL AND [ProductVersionId] IS NOT NULL) OR
        ([InstallationId] IS NOT NULL AND [ProductVersionId] IS NULL)");
    });
    model.Entity<EffectiveConfigurationWork>(b =>
    {
        b.HasIndex(x => new { x.InstallationId, x.ProductVersionId }).IsUnique();
        b.HasIndex(x => new { x.Status, x.NextAttemptAt });
        b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).onDelete(DeleteBehavior.NoAction);
        b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.ProductVersionId).onDelete(DeleteBehavior.NoAction);
    });
    model.Entity<EffectiveConfigurationRevision>(b =>
    {
        b.Property(x => x.InputFingerprint).HasMaxLength(64);
        b.HasOne<FileArtifact>().WithMany().HasForeignKey(x =>
x.SchemaArtifactId).onDelete(DeleteBehavior.NoAction);
        b.HasIndex(x => new { x.InstallationId, x.InputFingerprint
}).IsUnique().HasFilter("[InputFingerprint] IS NOT NULL");
    });
    }
}

```

FileArtifactConfiguration.cs

```

using BridgeControl.Models Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for FileArtifact records.
/// </summary>
public class FileArtifactConfiguration : IEntityTypeConfiguration<FileArtifact>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="FileArtifact"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<FileArtifact> builder)
    {
        builder.ToTable("FileArtifacts", table =>
            table.HasCheckConstraint("CK_FileArtifacts_Sha256",
                "DATALENGTH([Sha256]) = 64 AND [Sha256] COLLATE
Latin1_General_100_BIN2 NOT LIKE '%[^0-9A-Fa-f]%'");

        builder.Property(entity => entity.ArtifactType)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.FileName)
            .HasMaxLength(255)
            .IsRequired();

        builder.Property(entity => entity.StorageIdentifier)
            .HasMaxLength(1024)
            .IsRequired();

        builder.Property(entity => entity.ContentType)

```

```

        .HasMaxLength(255)
        .IsRequired();

        builder.Property(entity => entity.Sha256)
            .HasMaxLength(64)
            .IsUnicode(false)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);
    }
}

```

InstallationConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for Installation records.
/// </summary>
public class InstallationConfiguration : IEntityTypeConfiguration<Installation>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Installation"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<Installation> builder)
    {
        builder.ToTable("Installations");

        // The model initializes Id; it must not become a temporary key in the circular
        relationship.
        builder.Property(entity => entity.Id)
            .ValueGeneratedNever();

        builder.Property(entity => entity.Name)
            .HasMaxLength(200)
            .IsRequired();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.
        builder.HasOne(entity => entity.Customer)
            .WithMany(entity => entity.Installations)
            .HasForeignKey(entity => entity.CustomerId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.Product)
            .WithMany(entity => entity.Installations)
            .HasForeignKey(entity => entity.ProductId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.CurrentEffectiveConfigurationRevision)
            .WithMany(entity => entity.CurrentForInstallations)
            // Include this installation's ID so another installation's revision cannot
            become current.
    }
}

```

```

        .HasForeignKey(entity => new {
entity.CurrentEffectiveConfigurationRevisionId, entity.Id })
        .HasPrincipalKey(entity => new { entity.Id, entity.InstallationId })
        .IsRequired(false)
        // Do not automatically clear tracked optional links when the parent is
deleted.

        .OnDelete(DeleteBehavior.ClientNoAction);
    }
}

```

InstallationStateConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Data.Configurations.Vendor;

using Agent = BridgeControl.Models.AgentModels.Agent;

/// <summary>
/// Configures database mappings and constraints for installation state entities.
/// </summary>
public static class InstallationStateConfiguration
{
    /// <summary>
    /// Configures mappings and constraints for desired state, reports, reconciliation work,
and deployments.
    /// </summary>
    public static void Configure(ModelBuilder model)
    {
        model.Entity<DesiredStateRevision>(b =>
        {
            b.HasAlternateKey(x => new { x.Id, x.InstallationId });
            b.HasIndex(x => new { x.InstallationId, x.RevisionNumber }).IsUnique();
            b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.ProductVersionId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<EffectiveConfigurationRevision>().WithMany().HasForeignKey(x =>
new { x.EffectiveConfigurationRevisionId, x.InstallationId })
                .HasPrincipalKey(x => new { x.Id, x.InstallationId
}).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<UserAccount>().WithMany().HasForeignKey(x =>
x.CreatedByUserId).OnDelete(DeleteBehavior.NoAction);
            b.ToTable("DesiredStateRevisions", t => {
t.HasTrigger("TR_DesiredStateRevisions_Immutable"); t.UseSqlOutputClause(false); });
        });
        model.Entity<InstallationStateReport>(b =>
        {
            b.HasAlternateKey(x => new { x.Id, x.InstallationId });
            b.HasIndex(x => new { x.InstallationId, x.ReportedAt }).IsUnique();
            b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<Agent>().WithMany().HasForeignKey(x =>
x.AgentId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.InstalledProductVersionId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<EffectiveConfigurationRevision>().WithMany().HasForeignKey(x =>
new { x.EffectiveConfigurationRevisionId, x.InstallationId })

```

```

        .HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
    b.ToTable("InstallationStateReports", t => {
t.HasTrigger("TR_InstallationStateReports_Immutable"); t.UseSqlOutputClause(false); });
    });
    model.Entity<Installation>(b =>
    {
        b.HasOne<DesiredStateRevision>().WithMany().HasForeignKey(x => new {
x.DesiredStateRevisionId, x.Id }).HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
        b.HasOne<InstallationStateReport>().WithMany().HasForeignKey(x => new {
x.ReportedStateReportId, x.Id }).HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
    });
    model.Entity<StateReconciliationWork>(b =>
    {
        b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).onDelete(DeleteBehavior.NoAction);
        b.HasIndex(x => new { x.Status, x.NextAttemptAt });
    });
    model.Entity<Deployment>(b =>
    {
        b.HasAlternateKey(x => new { x.Id, x.InstallationId });
        b.HasIndex(x => x.InstallationId).IsUnique().HasFilter("[IsActive] = 1");
        b.HasIndex(x => new { x.InstallationId, x.ReconciliationGeneration
    }).IsUnique();
        b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).onDelete(DeleteBehavior.NoAction);
        b.HasOne<DesiredStateRevision>().WithMany().HasForeignKey(x => new {
x.DesiredStateRevisionId, x.InstallationId }).HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
        b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.TargetProductId).onDelete(DeleteBehavior.NoAction);
        b.HasOne<EffectiveConfigurationRevision>().WithMany().HasForeignKey(x =>
new { x.TargetConfigurationRevisionId, x.InstallationId }).HasPrincipalKey(x => new { x.Id,
x.InstallationId }).onDelete(DeleteBehavior.NoAction);
        b.HasOne<InstallationStateReport>().WithMany().HasForeignKey(x => new {
x.ReconciledReportId, x.InstallationId }).HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
    });
    model.Entity<DeploymentAttempt>(b =>
    {
        b.HasIndex(x => x.InstallationId).IsUnique().HasFilter("[CompletedAt] IS
NULL");
        b.HasOne<Agent>().WithMany().HasForeignKey(x =>
x.AgentId).onDelete(DeleteBehavior.NoAction);
        b.HasOne<Deployment>().WithMany().HasForeignKey(x => new { x.DeploymentId,
x.InstallationId }).HasPrincipalKey(x => new { x.Id, x.InstallationId
    }).onDelete(DeleteBehavior.NoAction);
        b.ToTable("DeploymentAttempts", t => {
t.HasTrigger("TR_DeploymentAttempts_ImmutableCompleted"); t.UseSqlOutputClause(false); });
        b.HasIndex(x => new { x.DeploymentId, x.AttemptNumber }).IsUnique();
    });
    }
}

```

LicenseConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

```

```

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for license entities.
/// </summary>
public static class LicenseConfiguration
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="License"/>.
    /// </summary>
    public static void Configure(ModelBuilder model)
    {
        model.Entity<License>(b =>
        {
            b.ToTable("Licenses", t =>
            {
                t.HasCheckConstraint("CK_Licenses_Validity", "[ValidUntil] IS NULL
OR [ValidUntil] > [ValidFrom]");
                t.HasCheckConstraint("CK_Licenses_Capacity", "[MaxInstallations] IS
NULL OR [MaxInstallations] > 0");
            });
            b.HasAlternateKey(x => new { x.Id, x.CustomerId, x.ProductId });
            b.HasOne(x => x.Customer).WithMany().HasForeignKey(x =>
x.CustomerId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne(x => x.Product).WithMany().HasForeignKey(x =>
x.ProductId).OnDelete(DeleteBehavior.NoAction);
        });
        model.Entity<InstallationLicenseAssignment>(b =>
        {
            b.HasOne(x => x.Installation).WithMany().HasForeignKey(x => new {
x.InstallationId, x.CustomerId, x.ProductId })
                .HasPrincipalKey(x => new { x.Id, x.CustomerId, x.ProductId
}).OnDelete(DeleteBehavior.NoAction);
            b.HasOne(x => x.License).WithMany().HasForeignKey(x => new { x.LicenseId,
x.CustomerId, x.ProductId })
                .HasPrincipalKey(x => new { x.Id, x.CustomerId, x.ProductId
}).OnDelete(DeleteBehavior.NoAction);
            b.HasIndex(x => x.InstallationId).IsUnique().HasFilter("[RemovedAt] IS
NULL");
            b.HasIndex(x => new { x.LicenseId, x.RemovedAt });
        });
    }
}

```

MigrationRuleConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for MigrationRule records.
/// </summary>
public class MigrationRuleConfiguration : IEntityTypeConfiguration<MigrationRule>
{
    /// <summary>

```

```

    /// Configures the table, properties, relationships, and constraints for <see
    cref="MigrationRule"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<MigrationRule> builder)
    {
        builder.ToTable("MigrationRules");

        builder.Property(entity => entity.RuleType)
            .HasMaxLength(64)
            .IsRequired();

        builder.Property(entity => entity.SourcePath)
            .HasMaxLength(2048)
            .IsRequired();

        builder.Property(entity => entity.TargetPath)
            .HasMaxLength(2048)
            .IsRequired();

        builder.Property(entity => entity.Transform)
            .HasColumnType("nvarchar(max)")
            .IsRequired();

        builder.HasIndex(entity => new { entity.MigrationId, entity.Sequence })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.Migration)
            .WithMany(entity => entity.Rules)
            .HasForeignKey(entity => entity.MigrationId)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

PasswordResetConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for password reset entities.
/// </summary>
public class PasswordResetConfiguration : IEntityTypeConfiguration<PasswordReset>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="PasswordReset"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<PasswordReset> builder)
    {
        builder.ToTable("PasswordResets", table =>
            table.HasCheckConstraint("CK_PasswordResets_Expiry", "[ExpiresAt] > [CreatedAt]"));
        builder.Property(x => x.TokenHash).HasMaxLength(32).IsFixedLength().IsRequired();
        builder.Property(x => x.ProtectedToken).HasMaxLength(2048);
        builder.Property(x => x.RowVersion).IsRowVersion();
    }
}

```

```

        builder.HasIndex(x => x.UserId).IsUnique().HasFilter("[UsedAt] IS NULL AND
[RevokedAt] IS NULL");
        builder.HasOne(x => x.User).WithMany().HasForeignKey(x =>
x.UserId).OnDelete(DeleteBehavior.NoAction);
    }
}

```

PermissionConfiguration.cs

```

using BridgeControl.Models Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for Permission records.
/// </summary>
public class PermissionConfiguration : IEntityTypeConfiguration<Permission>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Permission"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<Permission> builder)
    {
        builder.ToTable("Permissions");

        builder.Property(entity => entity.Code)
            .HasMaxLength(100)
            .IsRequired();

        builder.Property(entity => entity.Description)
            .HasMaxLength(2000)
            .IsRequired();
    }
}

```

ProductConfiguration.cs

```

using BridgeControl.Models Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for Product records.
/// </summary>
public class ProductConfiguration : IEntityTypeConfiguration<Product>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Product"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<Product> builder)
    {

```

```

        builder.ToTable("Products");
        builder.Property(x => x.RowVersion).IsRowVersion();

        builder.Property(entity => entity.Name)
            .HasMaxLength(200)
            .IsRequired();

        builder.Property(entity => entity.Description)
            .HasMaxLength(2000);

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);
    }
}

```

ProductVersionArtifactConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for ProductVersionArtifact records.
/// </summary>
public class ProductVersionArtifactConfiguration :
    IEntityTypeConfiguration<ProductVersionArtifact>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="ProductVersionArtifact"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<ProductVersionArtifact> builder)
    {
        builder.ToTable("ProductVersionArtifacts");
        builder.Property(x => x.ArtifactType).HasMaxLength(32).IsRequired();
        builder.HasIndex(x => new { x.ProductVersionId, x.ArtifactType }).IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.ProductVersion)
            .WithMany(entity => entity.Artifacts)
            .HasForeignKey(entity => entity.ProductVersionId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.FileArtifact)
            .WithMany(entity => entity.ProductVersions)
            .HasForeignKey(entity => new { entity.FileArtifactId, entity.ArtifactType })

        .HasPrincipalKey(entity => new { entity.Id, entity.ArtifactType })
        .OnDelete(DeleteBehavior.NoAction);
    }
}

```


ProductVersionConfiguration.cs

```
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for ProductVersion records.
/// </summary>
public class ProductVersionConfiguration : IEntityTypeConfiguration<ProductVersion>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="ProductVersion"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<ProductVersion> builder)
    {
        builder.ToTable("ProductVersions");
        builder.HasOne<FileArtifact>().WithMany().HasForeignKey(x =>
x.PublishedSchemaArtifactId).OnDelete(DeleteBehavior.NoAction);
        builder.HasOne<FileArtifact>().WithMany().HasForeignKey(x =>
x.PublishedInstallerArtifactId).OnDelete(DeleteBehavior.NoAction);
        builder.HasOne<ConfigurationRevision>().WithMany().HasForeignKey(x =>
x.PublishedDefaultRevisionId).OnDelete(DeleteBehavior.NoAction);
        builder.Property(x => x.RowVersion).IsRowVersion();

        builder.Property(entity => entity.Version)
            .HasMaxLength(100)
            .IsRequired();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.PublishedAt)
            .HasPrecision(7);

        builder.HasIndex(entity => new { entity.ProductId, entity.Version })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.Product)
            .WithMany(entity => entity.Versions)
            .HasForeignKey(entity => entity.ProductId)
            .OnDelete(DeleteBehavior.NoAction);
    }
}
```

RoleConfiguration.cs

```
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
```

```

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for Role records.
/// </summary>
public class RoleConfiguration : IEntityTypeConfiguration<Role>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="Role"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<Role> builder)
    {
        builder.ToTable("Roles");

        builder.Property(entity => entity.Name)
            .HasMaxLength(200)
            .IsRequired();

        builder.Property(entity => entity.Scope)
            .HasMaxLength(32)
            .IsRequired();
    }
}

```

RolePermissionConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for RolePermission records.
/// </summary>
public class RolePermissionConfiguration : IEntityTypeConfiguration<RolePermission>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="RolePermission"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<RolePermission> builder)
    {
        builder.ToTable("RolePermissions");

        builder.HasIndex(entity => new { entity.RoleId, entity.PermissionId })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.
        builder.HasOne(entity => entity.Role)
            .WithMany(entity => entity.RolePermissions)
            .HasForeignKey(entity => entity.RoleId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.Permission)
            .WithMany(entity => entity.RolePermissions)
            .HasForeignKey(entity => entity.PermissionId)

```

```

        .OnDelete(DeleteBehavior.NoAction);
    }
}

```

TechnicianCustomerAccessConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for TechnicianCustomerAccess records.
/// </summary>
public class TechnicianCustomerAccessConfiguration :
    IEntityTypeConfiguration<TechnicianCustomerAccess>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="TechnicianCustomerAccess"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<TechnicianCustomerAccess> builder)
    {
        builder.ToTable("TechnicianCustomerAccessGrants");

        builder.Property(entity => entity.GrantedAt)
            .HasPrecision(7);

        builder.HasIndex(entity => new { entity.UserId, entity.CustomerId })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.User)
            .WithMany(entity => entity.CustomerAccessGrants)
            .HasForeignKey(entity => entity.UserId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.Customer)
            .WithMany(entity => entity.TechnicianAccessGrants)
            .HasForeignKey(entity => entity.CustomerId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.GrantedByUser)
            .WithMany(entity => entity.GrantedCustomerAccess)
            .HasForeignKey(entity => entity.GrantedById)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

UserAccountConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

```

```

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage for UserAccount records.
/// </summary>
public class UserAccountConfiguration : IEntityTypeConfiguration<UserAccount>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="UserAccount"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<UserAccount> builder)
    {
        builder.ToTable("UserAccounts");
        builder.Property(entity => entity.RowVersion).IsRowVersion();

        builder.Property(entity => entity.Email)
            .UseCollation("Latin1_General_100_CI_AS_SC")
            .HasMaxLength(254)
            .IsRequired();

        builder.Property(entity => entity.PasswordHash)
            .HasMaxLength(512)
            .IsRequired();

        builder.Property(entity => entity.Status)
            .HasMaxLength(32)
            .IsRequired();

        builder.Property(entity => entity.CreatedAt)
            .HasPrecision(7);

        builder.Property(entity => entity.LastLoginAt)
            .HasPrecision(7);

        builder.HasIndex(entity => new { entity.Id, entity.CustomerId })
            .IsUnique().HasFilter(null);

        builder.HasMany(entity => entity.Sessions)
            .WithOne()
            .HasForeignKey(session => session.AccountId)
            .IsRequired()
            .OnDelete(DeleteBehavior.NoAction);

        // Preserve dependent records; deleting relationships must be an explicit
        operation.
        builder.HasOne(entity => entity.Customer)
            .WithMany(entity => entity.Users)
            .HasForeignKey(entity => entity.CustomerId)
            // Do not automatically clear tracked optional links when the parent is
            deleted.
            .OnDelete(DeleteBehavior.ClientNoAction);
    }
}

```

UserRoleConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

```

```

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Defines database storage and uniqueness rules for UserRole records.
/// </summary>
public class UserRoleConfiguration : IEntityTypeConfiguration<UserRole>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="UserRole"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<UserRole> builder)
    {
        builder.ToTable("UserRoles");

        builder.HasIndex(entity => new { entity.UserId, entity.RoleId })
            .IsUnique();

        // Preserve dependent records; deleting relationships must be an explicit
        operation.

        builder.HasOne(entity => entity.User)
            .WithMany(entity => entity.UserRoles)
            .HasForeignKey(entity => entity.UserId)
            .OnDelete(DeleteBehavior.NoAction);

        builder.HasOne(entity => entity.Role)
            .WithMany(entity => entity.UserRoles)
            .HasForeignKey(entity => entity.RoleId)
            .OnDelete(DeleteBehavior.NoAction);
    }
}

```

VendorAuditLogConfiguration.cs

```

using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace BridgeControl.Data.Configurations.Vendor;

/// <summary>
/// Configures database mappings and constraints for vendor audit log entities.
/// </summary>
public class VendorAuditLogConfiguration : IEntityTypeConfiguration<VendorAuditLog>
{
    /// <summary>
    /// Configures the table, properties, relationships, and constraints for <see
    cref="VendorAuditLog"/>.
    /// </summary>
    public void Configure(EntityTypeBuilder<VendorAuditLog> builder)
    {
        builder.Property(entity => entity.ActorType).HasMaxLength(32).IsRequired();
        builder.Property(entity => entity.Action).HasMaxLength(100).IsRequired();
        builder.Property(entity => entity.EntityType).HasMaxLength(64).IsRequired();
        builder.Property(entity => entity.Outcome).HasMaxLength(32).IsRequired();
        builder.Property(entity => entity.Context).HasMaxLength(512).IsRequired();
        builder.HasIndex(entity => new { entity.EntityId, entity.CreatedAt });
        builder.HasIndex(entity => new { entity.CreatedAt, entity.Id });
    }
}

```

```
}
```

Data/Factories

VendorDbContextFactory.cs

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;

namespace BridgeControl.Data.Factories;

/// <summary>
/// Creates the development vendor context for EF migration commands.
/// </summary>
public class VendorDbContextFactory
    : IDesignTimeDbContextFactory<VendorDbContext>
{
    /// <summary>
    /// Creates a vendor database context for Entity Framework design-time tools.
    /// </summary>
    public VendorDbContext CreateDbContext(string[] args)
    {
        IConfiguration configuration = new ConfigurationBuilder()
            .AddUserSecrets<VendorDbContextFactory>()
            .AddEnvironmentVariables()
            .Build();

        string connectionString =
            configuration.GetConnectionString("DevelopmentVendorDatabase")
            ?? throw new InvalidOperationException(
                "The development vendor database connection string is missing.");

        DbContextOptionsBuilder<VendorDbContext> options = new();
        options.UseSqlServer(connectionString);

        return new VendorDbContext(options.Options);
    }
}
```

Data

PlatformDbContext.cs

```
using BridgeControl.Data.Configurations.Platform;
using BridgeControl.Data.Configurations.Shared;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Data;
```

```

/// <summary>
/// Provides database access to platform administrators, vendors and platform audit records.
/// </summary>
public class PlatformDbContext(DbContextOptions<PlatformDbContext> options) : DbContext(options)
{
    public DbSet<VendorMaintenanceOperation> VendorMaintenanceOperations =>
Set<VendorMaintenanceOperation>();
    public DbSet<PlatformAdmin> PlatformAdmins => Set<PlatformAdmin>();

    public DbSet<SoftwareVendor> SoftwareVendors => Set<SoftwareVendor>();

    public DbSet<VendorDatabase> VendorDatabases => Set<VendorDatabase>();

    public DbSet<VendorFileArchive> VendorFileArchives => Set<VendorFileArchive>();

    public DbSet<PlatformAuditLog> PlatformAuditLogs => Set<PlatformAuditLog>();

    public DbSet<AuthenticationSession> AuthenticationSessions =>
Set<AuthenticationSession>();

    public DbSet<RefreshToken> RefreshTokens => Set<RefreshToken>();

    public DbSet<VendorProvisioning> VendorProvisioning => Set<VendorProvisioning>();

    /// <summary>
    /// Applies entity mappings, relationships, and database constraints to the model.
    /// </summary>
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Register only this database's configurations; both contexts share one assembly.
        modelBuilder.ApplyConfiguration(new PlatformAdminConfiguration());
        modelBuilder.ApplyConfiguration(new VendorMaintenanceOperationConfiguration());
        modelBuilder.ApplyConfiguration(new SoftwareVendorConfiguration());
        modelBuilder.ApplyConfiguration(new VendorProvisioningConfiguration());
        modelBuilder.ApplyConfiguration(new VendorDatabaseConfiguration());
        modelBuilder.ApplyConfiguration(new VendorFileArchiveConfiguration());
        modelBuilder.ApplyConfiguration(new PlatformAuditLogConfiguration());
        modelBuilder.ApplyConfiguration(new AuthenticationSessionConfiguration());
        modelBuilder.ApplyConfiguration(new RefreshTokenConfiguration());
    }
}

```

VendorDbContext.cs

```

using BridgeControl.Data.Configurations.AgentConfigurations;
using BridgeControl.Data.Configurations.Shared;
using BridgeControl.Data.Configurations.Vendor;
using BridgeControl.Models.AgentModels;
using BridgeControl.Models.Shared;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Data;

/// <summary>
/// Provides access to one vendor's customers, users, products and configurations.
/// </summary>

```

```

public class VendorDbContext(DbContextOptions<VendorDbContext> options) : DbContext(options)
{
    public DbSet<OperationalLog> OperationalLogs => Set<OperationalLog>();
    public DbSet<LogIngestionWindow> LogIngestionWindows => Set<LogIngestionWindow>();
    public DbSet<DesiredStateRevision> DesiredStateRevisions => Set<DesiredStateRevision>();
    public DbSet<InstallationStateReport> InstallationStateReports =>
Set<InstallationStateReport>();
    public DbSet<StateReconciliationWork> StateReconciliationWork =>
Set<StateReconciliationWork>();
    public DbSet<Deployment> Deployments => Set<Deployment>();
    public DbSet<DeploymentAttempt> DeploymentAttempts => Set<DeploymentAttempt>();
    public DbSet<Agent> Agents => Set<Agent>();
    public DbSet<License> Licenses => Set<License>();
    public DbSet<InstallationLicenseAssignment> InstallationLicenseAssignments =>
Set<InstallationLicenseAssignment>();
    public DbSet<AgentEnrollment> AgentEnrollments => Set<AgentEnrollment>();
    public DbSet<AgentCredential> AgentCredentials => Set<AgentCredential>();
    public DbSet<AgentHeartbeat> AgentHeartbeats => Set<AgentHeartbeat>();
    public DbSet<Customer> Customers => Set<Customer>();

    public DbSet<UserAccount> UserAccounts => Set<UserAccount>();

    public DbSet<Role> Roles => Set<Role>();

    public DbSet<Permission> Permissions => Set<Permission>();

    public DbSet<UserRole> UserRoles => Set<UserRole>();

    public DbSet<RolePermission> RolePermissions => Set<RolePermission>();

    public DbSet<TechnicianCustomerAccess> TechnicianCustomerAccessGrants
=> Set<TechnicianCustomerAccess>();

    public DbSet<Product> Products => Set<Product>();

    public DbSet<ProductVersion> ProductVersions => Set<ProductVersion>();

    public DbSet<FileArtifact> FileArtifacts => Set<FileArtifact>();

    public DbSet<ProductVersionArtifact> ProductVersionArtifacts
=> Set<ProductVersionArtifact>();

    public DbSet<Installation> Installations => Set<Installation>();

    public DbSet<ConfigurationProfile> ConfigurationProfiles
=> Set<ConfigurationProfile>();

    public DbSet<ConfigurationRevision> ConfigurationRevisions
=> Set<ConfigurationRevision>();
    public DbSet<ConfigurationSourceChange> ConfigurationSourceChanges =>
Set<ConfigurationSourceChange>();
    public DbSet<EffectiveConfigurationWork> EffectiveConfigurationWork =>
Set<EffectiveConfigurationWork>();
    public DbSet<EffectiveConfigurationSignal> EffectiveConfigurationSignals =>
Set<EffectiveConfigurationSignal>();

    public DbSet<EffectiveConfigurationRevision> EffectiveConfigurationRevisions
=> Set<EffectiveConfigurationRevision>();

    public DbSet<EffectiveConfigurationSourceRevision> EffectiveConfigurationSourceRevisions
=> Set<EffectiveConfigurationSourceRevision>();

```



```

    public DbSet<ConfigurationMigration> ConfigurationMigrations
        => Set<ConfigurationMigration>();

    public DbSet<MigrationRule> MigrationRules => Set<MigrationRule>();
    public DbSet<MigrationDefinitionRevision> MigrationDefinitionRevisions =>
Set<MigrationDefinitionRevision>();
    public DbSet<ConfigurationMigrationReview> ConfigurationMigrationReviews =>
Set<ConfigurationMigrationReview>();

    public DbSet<AuthenticationSession> AuthenticationSessions =>
Set<AuthenticationSession>();

    public DbSet<RefreshToken> RefreshTokens => Set<RefreshToken>();

    public DbSet<CustomerUserInstallationAccess> CustomerUserInstallationAccessGrants =>
Set<CustomerUserInstallationAccess>();

    public DbSet<AccountActivation> AccountActivations => Set<AccountActivation>();

    public DbSet<VendorAuditLog> VendorAuditLogs => Set<VendorAuditLog>();
    public DbSet<PasswordReset> PasswordResets => Set<PasswordReset>();

    /// <summary>
    /// Applies entity mappings, relationships, and database constraints to the model.
    /// </summary>
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        modelBuilder.Entity<OperationalLog>(b =>
        {
            b.HasIndex(x => new { x.AgentId, x.Fingerprint }).IsUnique();
            b.HasIndex(x => new { x.InstallationId, x.CreatedAt, x.Id });
            b.HasIndex(x => new { x.AgentId, x.CreatedAt, x.Id });
            b.HasIndex(x => x.ReceivedAt);
            b.HasOne<Agent>().WithMany().HasForeignKey(x =>
x.AgentId).onDelete(DeleteBehavior.NoAction);
            b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).onDelete(DeleteBehavior.NoAction);
        });

        modelBuilder.Entity<LogIngestionWindow>().HasOne<Agent>().WithMany().HasForeignKey(x =>
x.AgentId).onDelete(DeleteBehavior.NoAction);
        InstallationStateConfiguration.Configure(modelBuilder);
        AgentConfiguration.Configure(modelBuilder);
        LicenseConfiguration.Configure(modelBuilder);
        EffectiveConfigurationWorkConfiguration.Configure(modelBuilder);

        // Register only this database's configurations; both contexts share one assembly.
        modelBuilder.ApplyConfiguration(new CustomerConfiguration());
        modelBuilder.ApplyConfiguration(new UserAccountConfiguration());
        modelBuilder.ApplyConfiguration(new RoleConfiguration());
        modelBuilder.ApplyConfiguration(new PermissionConfiguration());
        modelBuilder.ApplyConfiguration(new UserRoleConfiguration());
        modelBuilder.ApplyConfiguration(new RolePermissionConfiguration());
        modelBuilder.ApplyConfiguration(new TechnicianCustomerAccessConfiguration());
        modelBuilder.ApplyConfiguration(new ProductConfiguration());
        modelBuilder.ApplyConfiguration(new ProductVersionConfiguration());
        modelBuilder.ApplyConfiguration(new FileArtifactConfiguration());
        modelBuilder.ApplyConfiguration(new ProductVersionArtifactConfiguration());
        modelBuilder.ApplyConfiguration(new InstallationConfiguration());
        modelBuilder.ApplyConfiguration(new ConfigurationProfileConfiguration());
        modelBuilder.ApplyConfiguration(new ConfigurationRevisionConfiguration());

```

```

        modelBuilder.Entity<ConfigurationSourceChange>(b =>
        {
            b.HasOne<ConfigurationRevision>().WithMany().HasForeignKey(x =>
x.ConfigurationRevisionId).OnDelete(DeleteBehavior.NoAction);
            b.HasIndex(x => x.ConfigurationRevisionId).IsUnique();
            b.HasIndex(x => new { x.ProcessedAt, x.CreatedAt });
        });
        modelBuilder.ApplyConfiguration(new
EffectiveConfigurationRevisionConfiguration());
        modelBuilder.ApplyConfiguration(new
EffectiveConfigurationSourceRevisionConfiguration());
        modelBuilder.ApplyConfiguration(new ConfigurationMigrationConfiguration());
        modelBuilder.ApplyConfiguration(new MigrationRuleConfiguration());
        modelBuilder.Entity<MigrationDefinitionRevision>(b =>
        {
            b.HasIndex(x => new { x.MigrationId, x.RevisionNumber }).IsUnique();
            b.HasOne<ConfigurationMigration>().WithMany().HasForeignKey(x =>
x.MigrationId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<UserAccount>().WithMany().HasForeignKey(x =>
x.CreatedByUserId).OnDelete(DeleteBehavior.NoAction);
            b.ToTable("MigrationDefinitionRevisions", t => {
t.HasTrigger("TR_MigrationDefinitionRevisions_Immutable"); t.UseSqlOutputClause(false); });
        });
        modelBuilder.Entity<ConfigurationMigrationReview>(b =>
        {
            b.ToTable("ConfigurationMigrationReviews", t => {
t.HasTrigger("TR_ConfigurationMigrationReviews_Immutable"); t.UseSqlOutputClause(false); });
            b.HasIndex(x => new { x.UserId, x.InstallationId, x.TargetProductVersionId,
x.CreatedAt });
            b.HasOne<UserAccount>().WithMany().HasForeignKey(x =>
x.UserId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<Installation>().WithMany().HasForeignKey(x =>
x.InstallationId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<ProductVersion>().WithMany().HasForeignKey(x =>
x.TargetProductVersionId).OnDelete(DeleteBehavior.NoAction);
            b.HasOne<EffectiveConfigurationRevision>().WithMany().HasForeignKey(x =>
x.ResultRevisionId).OnDelete(DeleteBehavior.NoAction);
        });
        modelBuilder.ApplyConfiguration(new AuthenticationSessionConfiguration());
        modelBuilder.ApplyConfiguration(new RefreshTokenConfiguration());
        modelBuilder.ApplyConfiguration(new
CustomerUserInstallationAccessConfiguration());
        modelBuilder.ApplyConfiguration(new AccountActivationConfiguration());
        modelBuilder.ApplyConfiguration(new VendorAuditLogConfiguration());
        modelBuilder.ApplyConfiguration(new PasswordResetConfiguration());
    }
}

```

DTOs/AgentDTOs/AgentContracts

AgentEnrollmentResponse.cs

```

namespace BridgeControl.DTOs.AgentDTOs.AgentContracts;

/// <summary>
/// Carries agent enrollment data returned by the API.

```

```
/// </summary>
public sealed record AgentEnrollmentResponse(string EnrollmentCode, DateTimeOffset ExpiresAt);
```

AgentHeartbeatRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries input for the agent heartbeat operation.
/// </summary>
public sealed class AgentHeartbeatRequest
{
    [Required, MaxLength(100)] public string AgentVersion { get; set; } = "";
    [Required, MaxLength(100)] public string Status { get; set; } = "";
    public DateTimeOffset ReportedAt { get; set; }
}
```

AgentHeartbeatResponse.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries agent heartbeat data returned by the API.
/// </summary>
public sealed record AgentHeartbeatResponse(string AgentVersion, string Status, DateTimeOffset ReportedAt, DateTimeOffset ReceivedAt);
```

AgentRegistrationResponse.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries agent registration data returned by the API.
/// </summary>
public sealed record AgentRegistrationResponse(Guid AgentId, string Credential, DateTimeOffset ExpiresAt);
```

AgentResponse.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries agent data returned by the API.
/// </summary>
public sealed record AgentResponse(Guid Id, Guid CustomerId, string MachineName, string AgentVersion,
    string Status, string? ReportedStatus, DateTimeOffset RegisteredAt, DateTimeOffset? LastSeenAt);
```

AgentTokenRequest.cs

```
using System.ComponentModel.DataAnnotations;
using BridgeControl.Api;

namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries input for the agent token operation.
/// </summary>
public sealed class AgentTokenRequest { [Required, MaxLength(160), PreserveWhitespace] public
string Credential { get; set; } = ""; }
```

AgentTokenResponse.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries agent token data returned by the API.
/// </summary>
public sealed record AgentTokenResponse(string AccessToken, DateTimeOffset ExpiresAt);
```

CreateAgentEnrollmentRequest.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries input for the create agent enrollment operation.
/// </summary>
public sealed class CreateAgentEnrollmentRequest { public Guid CustomerId { get; set; } }
```

InstallationResponse.cs

```
namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries installation data returned by the API.
/// </summary>
public sealed record InstallationResponse(Guid Id, Guid CustomerId, Guid ProductId, Guid?
AgentId,
    string Name, string Status, DateTimeOffset RegisteredAt, DateTimeOffset? LastSeenAt);
```

RegisterAgentRequest.cs

```
using System.ComponentModel.DataAnnotations;
using BridgeControl.Api;

namespace BridgeControl.DTOS.AgentDTOS.AgentContracts;

/// <summary>
/// Carries input for the register agent operation.
/// </summary>
```

```

public sealed class RegisterAgentRequest
{
    [Required, MaxLength(160), PreserveWhitespace] public string EnrollmentCode { get; set; }
    = "";
    [Required, MaxLength(200)] public string MachineName { get; set; } = "";
    [Required, MaxLength(100)] public string AgentVersion { get; set; } = "";
}

```

RegisterInstallationRequest.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.AgentDTOs.AgentContracts;

/// <summary>
/// Carries input for the register installation operation.
/// </summary>
public sealed class RegisterInstallationRequest
{
    public Guid ProductId { get; set; }
    [Required, MaxLength(200)] public string Name { get; set; } = "";
    [MaxLength(100)] public string? InstalledVersion { get; set; }
}

```

DTOs/AgentDTOs/DeploymentSettings

AgentApplication.cs

```

namespace BridgeControl.DTOs.AgentDTOs.DeploymentSettings;

/// <summary>
/// Represents agent application data.
/// </summary>
public sealed record AgentApplication(Guid ProductId, Guid SettingsVersionId, string Name, string
Kind, Guid UpgradeCode,
    string ConfigurationPath, string ExecutablePath, string ServiceName,
    IReadOnlyList<AgentVersionBinding> Versions);

```

AgentInstallationAssignment.cs

```

namespace BridgeControl.DTOs.AgentDTOs.DeploymentSettings;

/// <summary>
/// Represents agent installation assignment data.
/// </summary>
public sealed record AgentInstallationAssignment(Guid Id, Guid CustomerId, Guid ProductId, Guid?
AgentId,
    string Name, string Status, DateTimeOffset RegisteredAt, DateTimeOffset? LastSeenAt,
    AgentApplication? Application);

```

AgentVersionBinding.cs

```
namespace BridgeControl.DTOS.AgentDTOS.DeploymentSettings;

/// <summary>
/// Represents agent version binding data.
/// </summary>
public sealed record AgentVersionBinding(Guid ProductVersionId, Guid ProductCode, string
Version);
```

DTOs/AgentDTOs/OperationalHistoryContr acts

AgentLogEntryRequest.cs

```
using System.ComponentModel.DataAnnotations;
namespace BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;

/// <summary>
/// Carries input for the agent log entry operation.
/// </summary>
public sealed record AgentLogEntryRequest(Guid? InstallationId, [Required, MaxLength(16)] string
Severity,
    [Required, MaxLength(100)] string Source, [Required, MaxLength(2000)] string Message,
    DateTimeOffset CreatedAt);
```

SubmitAgentLogsRequest.cs

```
using System.ComponentModel.DataAnnotations;
namespace BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;

/// <summary>
/// Carries input for the submit agent logs operation.
/// </summary>
public sealed record SubmitAgentLogsRequest([Required, MinLength(1), MaxLength(100)]
IEnumerable<AgentLogEntryRequest> Entries);
```

DTOs/Platform

CreateVendorAdministratorRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOS.Platform;
```

```

/// <summary>
/// Identifies the first vendor administrator to invite, without accepting a password.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateVendorAdministratorRequest
{
    [Required, EmailAddress, StringLength(254)]
    public required string Email { get; set; }
}

```

CreateVendorRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Supplies the name needed to register a software vendor on the platform.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateVendorRequest
{
    private string name = string.Empty;

    [Required]
    [StringLength(200)]
    public required string Name
    {
        get => name;
        set => name = value?.Trim() ?? string.Empty;
    }
}

```

PlatformAdminDto.cs

```

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Returns platform administrator account details without exposing the password hash.
/// </summary>
public class PlatformAdminDto
{
    public Guid Id { get; set; }

    public string Email { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? LastLoginAt { get; set; }
}

```

PlatformAuditLogDto.cs

```

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Returns a platform audit event with its actor, affected entity, optional vendor and
timestamp.
/// </summary>
public class PlatformAuditLogDto
{
    public Guid Id { get; set; }

    public Guid PlatformAdminId { get; set; }

    public Guid? VendorId { get; set; }

    public string Action { get; set; } = string.Empty;

    public string EntityType { get; set; } = string.Empty;

    public string EntityId { get; set; } = string.Empty;

    public string Details { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }
}

```

PlatformAuthSessionResponse.cs

```

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Returns the signed-in platform administrator and their short-lived access token.
/// </summary>
public class PlatformAuthSessionResponse
{
    public Guid PlatformAdminId { get; set; }
    public string Email { get; set; } = string.Empty;
    public string AccessToken { get; set; } = string.Empty;
    public DateTimeOffset AccessTokenExpiresAt { get; set; }
    public IReadOnlyList<string> Roles { get; set; } = [];
}

```

PlatformLoginRequest.cs

```

using System.ComponentModel.DataAnnotations;
using BridgeControl.Api;
using BridgeControl.Services.Shared;

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Supplies a platform administrator's email and password for login.
/// </summary>
public class PlatformLoginRequest
{
    [Required, EmailAddress, StringLength(254)]
    public string Email { get; set; } = string.Empty;
}

```



```

        [Required, StringLength(AccountValidation.MaximumPasswordLength), PreserveWhitespace]
        public string Password { get; set; } = string.Empty;
    }

```

UpdateVendorRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOS.Platform;

/// <summary>
/// Supplies an optional vendor name change; lifecycle changes use their own documented
/// operations.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateVendorRequest : IValidatableObject
{
    private string? name;

    [StringLength(200)]
    public string? Name
    {
        get => name;
        set { name = value?.Trim(); NameProvided = true; }
    }

    [JsonIgnore]
    public bool NameProvided { get; private set; }

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        if (NameProvided && string.IsNullOrEmpty(Name))
        {
            yield return new ValidationResult("A supplied name cannot be empty.",
[nameof(Name)]);
        }
    }
}

```

VendorDatabaseDto.cs

```

namespace BridgeControl.DTOS.Platform;

/// <summary>
/// Returns a vendor database record's identity and status without exposing connection details.
/// </summary>
public class VendorDatabaseDto
{
    public Guid Id { get; set; }

    public Guid VendorId { get; set; }

    public string Status { get; set; } = string.Empty;
}

```

```

        public DateTimeOffset CreatedAt { get; set; }
    }

```

VendorDto.cs

```

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Returns a software vendor's identity, name, status and lifecycle dates.
/// </summary>
public class VendorDto
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? DeactivatedAt { get; set; }
}

```

VendorFileArchiveDto.cs

```

namespace BridgeControl.DTOs.Platform;

/// <summary>
/// Returns a vendor file archive record's identity and status without exposing storage access
/// details.
/// </summary>
public class VendorFileArchiveDto
{
    public Guid Id { get; set; }

    public Guid VendorId { get; set; }

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }
}

```

DTOs/Shared

AuthenticationSessionDto.cs

```

namespace BridgeControl.DTOs.Shared;

/// <summary>
/// Returns a session's owner, expiry and revocation details without refresh credentials or
/// hashes.

```

```

/// </summary>
public class AuthenticationSessionDto
{
    public Guid Id { get; set; }

    public Guid AccountId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset ExpiresAt { get; set; }

    public DateTimeOffset? LastRefreshedAt { get; set; }

    public DateTimeOffset? RevokedAt { get; set; }
}

```

PagedResponse.cs

```

namespace BridgeControl.DTOs.Shared;

/// <summary>
/// Returns one page of results and pagination details for platform or vendor endpoints.
/// </summary>
public class PagedResponse<T>
{
    public IReadOnlyList<T> Items { get; set; } = [];

    public int Page { get; set; }

    public int PageSize { get; set; }

    public int TotalCount { get; set; }
}

```

PaginationQuery.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Shared;

/// <summary>
/// Validates bounded page requests before applying them to an authorized, consistently ordered
/// query.
/// </summary>
public sealed class PaginationQuery : IValidatableObject
{
    [Range(1, int.MaxValue)]
    public int Page { get; set; } = 1;

    [Range(1, 100)]
    public int PageSize { get; set; } = 25;

    internal int Offset => checked((Page - 1) * PageSize);

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
}

```

```

        public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
        {
            if (((long)Page - 1) * PageSize > int.MaxValue)
            {
                yield return new ValidationResult("The requested page is too large.",
[nameof(Page)]);
            }
        }
    }
}

```

DTOs/Vendor

ActivateAccountRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;
using BridgeControl.Api;
using BridgeControl.Services.Shared;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Exchanges a private activation token for the invited user's chosen password.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class ActivateAccountRequest
{
    [Required, StringLength(128), PreserveWhitespace]
    public required string Token { get; set; }

    [Required, PreserveWhitespace, DataType(DataType.Password)]
    [StringLength(AccountValidation.MaximumPasswordLength, MinimumLength =
AccountValidation.MinimumPasswordLength)]
    public required string Password { get; set; }
}

```

AssignRolePermissionRequest.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Identifies the permission to grant to a role.
/// </summary>
public class AssignRolePermissionRequest
{
    public required Guid RoleId { get; set; }

    public required Guid PermissionId { get; set; }
}

```

AssignUserRoleRequest.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Identifies the role to assign to a user account.
/// </summary>
public class AssignUserRoleRequest
{
    public required Guid UserId { get; set; }

    public required Guid RoleId { get; set; }
}

```

AttachProductVersionArtifactRequest.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Identifies an existing file to associate with a product version.
/// </summary>
public class AttachProductVersionArtifactRequest
{
    public required Guid ProductVersionId { get; set; }

    public required Guid FileArtifactId { get; set; }
}

```

ConfigurationMigrationDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a migration's source and target version IDs, author and creation time.
/// </summary>
public class ConfigurationMigrationDto
{
    public Guid Id { get; set; }

    public Guid FromProductVersionId { get; set; }

    public Guid ToProductVersionId { get; set; }

    public Guid CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    public IReadOnlyList<MigrationRuleDto> Rules { get; set; } = [];
}

```

DTOs/Vendor/ConfigurationMigrationExecution

ApplyConfigurationMigrationRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;

/// <summary>
/// Carries input for the apply configuration migration operation.
/// </summary>
public sealed record ApplyConfigurationMigrationRequest(Guid TargetProductId, [property:
Required] IReadOnlyList<Guid> ApprovedDestructiveRuleIds);
```

ConfigurationMigrationChangeResponse.cs

```
namespace BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;

/// <summary>
/// Carries configuration migration change data returned by the API.
/// </summary>
public sealed record ConfigurationMigrationChangeResponse(Guid RuleId, string Layer, string
RuleType, string SourcePath,
    string TargetPath, bool IsDestructive, string? PreviousValueHash);
```

ConfigurationMigrationPreviewRequest.cs

```
namespace BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;

/// <summary>
/// Carries input for the configuration migration preview operation.
/// </summary>
public sealed record ConfigurationMigrationPreviewRequest(Guid TargetProductId);
```

ConfigurationMigrationPreviewResponse.cs

```
using System.Text.Json;

namespace BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;

/// <summary>
/// Carries configuration migration preview data returned by the API.
/// </summary>
public sealed record ConfigurationMigrationPreviewResponse(Guid TargetProductId,
JsonElement Values, bool IsValid,
    IReadOnlyList<string> ValidationErrors,
    IReadOnlyList<ConfigurationMigrationChangeResponse> Changes);
```

DTOs/Vendor

ConfigurationProfileDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a configuration profile's identity and product, customer or installation scope.
/// </summary>
public class ConfigurationProfileDto
{
    public Guid Id { get; set; }

    public Guid? ProductId { get; set; }

    public Guid? CustomerId { get; set; }

    public Guid? InstallationId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }
}

```

ConfigurationRevisionDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a configuration revision's number, file and version references, author and creation
time.
/// </summary>
public class ConfigurationRevisionDto
{
    public Guid Id { get; set; }

    public Guid ConfigurationProfileId { get; set; }

    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }

    public int RevisionNumber { get; set; }

    public Guid CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }
}

```

ConfigurationRevisionResponse.cs

```

using System.Text.Json;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Carries configuration revision data returned by the API.
/// </summary>
public sealed record ConfigurationRevisionResponse(Guid Id, Guid ProductVersionId, int
RevisionNumber,
    JsonElement Values, Guid CreatedByUserId, DateTimeOffset CreatedAt);

```

CreateConfigurationMigrationRequest.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Identifies the source and target product versions for a new configuration migration.
/// </summary>
public class CreateConfigurationMigrationRequest
{
    public required Guid FromProductVersionId { get; set; }

    public required Guid ToProductVersionId { get; set; }

    [System.ComponentModel.DataAnnotations.Required]
    public required IReadOnlyList<MigrationRuleRequest> Rules { get; set; }
}
```

CreateConfigurationProfileRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Creates product defaults, customer overrides for a product, or installation overrides.
/// </summary>
public class CreateConfigurationProfileRequest : IValidatableObject
{
    public Guid? ProductId { get; set; }

    public Guid? CustomerId { get; set; }

    public Guid? InstallationId { get; set; }

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
    {
        bool isProductOrCustomerScope = ProductId.HasValue && !InstallationId.HasValue;
        bool isInstallationScope = InstallationId.HasValue && !ProductId.HasValue &&
!CustomerId.HasValue;

        if (!isProductOrCustomerScope && !isInstallationScope)
        {
            yield return new ValidationResult(
                "Specify ProductId for product defaults, ProductId and CustomerId
for customer overrides, " +
                "or InstallationId alone for installation overrides.",
                [nameof(ProductId), nameof(CustomerId), nameof(InstallationId)]);
        }
    }
}
```

CreateConfigurationRevisionRequest.cs

```
namespace BridgeControl.DTOs.Vendor;
```



```

/// <summary>
/// Identifies the profile, product version and stored file for a new configuration revision.
/// </summary>
public class CreateConfigurationRevisionRequest
{
    public required Guid ConfigurationProfileId { get; set; }

    public required Guid ProductVersionId { get; set; }

    public required Guid FileArtifactId { get; set; }
}

```

CreateCustomerRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the name needed to create a customer within the current vendor.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateCustomerRequest
{
    [Required, StringLength(200)]
    public required string Name { get; set; }
}

```

CreateInstallationRequest.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the customer, product and name for a new installation.
/// </summary>
public class CreateInstallationRequest
{
    public required Guid CustomerId { get; set; }

    public required Guid ProductId { get; set; }

    [Required]
    public required string Name { get; set; }
}

```

CreateMigrationRuleRequest.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor;

```

```

/// <summary>
/// Supplies an ordered migration step, including its operation, paths, transformation and
destructive flag.
/// </summary>
public class CreateMigrationRuleRequest
{
    public required Guid MigrationId { get; set; }

    public required int Sequence { get; set; }

    [Required]
    public required string RuleType { get; set; }

    [Required]
    public required string SourcePath { get; set; }

    [Required]
    public required string TargetPath { get; set; }

    [Required]
    public required string Transform { get; set; }

    public required bool IsDestructive { get; set; }
}

```

CreatePermissionRequest.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the code and description for a new permission.
/// </summary>
public class CreatePermissionRequest
{
    [Required]
    public required string Code { get; set; }

    [Required]
    public required string Description { get; set; }
}

```

CreateProductRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the name and description for a new software product.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateProductRequest
{
    [Required, StringLength(200)]

```

```

        public required string Name { get; set; }

        [StringLength(2000)]
        public string? Description { get; set; }
    }

```

CreateProductVersionRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the product and version label for a new release.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateProductVersionRequest
{
    [Required, StringLength(100)]
    public required string Version { get; set; }
}

```

CreateRoleRequest.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies the name and scope for a new access role.
/// </summary>
public class CreateRoleRequest
{
    [Required]
    public required string Name { get; set; }

    [Required]
    public required string Scope { get; set; }
}

```

CustomerDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a customer's identity, name, status and lifecycle dates.
/// </summary>
public class CustomerDto
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;
}

```

```

        public DateTimeOffset CreatedAt { get; set; }

        public DateTimeOffset? DeactivatedAt { get; set; }
    }

```

DTOs/Vendor/DeploymentContracts

DeploymentAttemptResponse.cs

```

namespace BridgeControl.DTOs.Vendor.DeploymentContracts;

/// <summary>
/// Carries deployment attempt data returned by the API.
/// </summary>
public sealed record DeploymentAttemptResponse(Guid Id, int AttemptNumber, string Status, string?
    ErrorCode,
    string? ErrorMessage, DateTimeOffset StartedAt, DateTimeOffset? CompletedAt);

```

DeploymentDetailResponse.cs

```

namespace BridgeControl.DTOs.Vendor.DeploymentContracts;

/// <summary>
/// Carries deployment detail data returned by the API.
/// </summary>
public sealed record DeploymentDetailResponse(Guid Id, Guid InstallationId, Guid
    TargetProductVersionId,
    Guid? TargetConfigurationRevisionId, string OperationType, string Status, DateTimeOffset
    RequestedAt, DateTimeOffset? CompletedAt,
    IReadOnlyList<DeploymentAttemptResponse> Attempts);

```

DeploymentResponse.cs

```

namespace BridgeControl.DTOs.Vendor.DeploymentContracts;

/// <summary>
/// Carries deployment data returned by the API.
/// </summary>
public record DeploymentResponse(Guid Id, Guid InstallationId, Guid TargetProductVersionId,
    Guid? TargetConfigurationRevisionId, string OperationType, string Status, DateTimeOffset
    RequestedAt, DateTimeOffset? CompletedAt);

```

StartDeploymentAttemptResponse.cs

```

namespace BridgeControl.DTOs.Vendor.DeploymentContracts;

/// <summary>
/// Carries start deployment attempt data returned by the API.
/// </summary>

```

```
public sealed record StartDeploymentAttemptResponse(Guid AttemptId, int AttemptNumber);
```

UpdateDeploymentAttemptRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.DeploymentContracts;

/// <summary>
/// Carries input for the update deployment attempt operation.
/// </summary>
public sealed record UpdateDeploymentAttemptRequest([Required, MaxLength(32)] string Status,
    [MaxLength(100)] string? ErrorCode, [MaxLength(2000)] string? ErrorMessage,
    DateTimeOffset? CompletedAt);
```

DTOs/Vendor/DeploymentSettings

AssignProductRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.DeploymentSettings;

/// <summary>
/// Carries input for the assign product operation.
/// </summary>
public sealed class AssignProductRequest
{
    public Guid AgentId { get; set; }
    public Guid ProductId { get; set; }
    [Required, MaxLength(200)] public string Name { get; set; } = "";
}
```

DeploymentSettings.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.DeploymentSettings;

/// <summary>
/// Represents deployment settings data.
/// </summary>
public sealed record DeploymentSettings(
    [property: Required] string Kind,
    [property: Required, MaxLength(500)] string ConfigurationPath,
    [property: MaxLength(500)] string? ExecutablePath = null,
    [property: MaxLength(200)] string? ServiceName = null) : IValidatableObject
{
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
```

```

{
    bool PathValid(string? path) => path is { Length: >= 4 and <= 500 } &&
char.IsAsciiLetter(path[0])
    && path[1] == ':' && path[2] == '\\' && !path[2..].Contains(':')
    && !path.Contains('%') && !path.Contains('/') && !path.Split('\\').Any(x =>
x is ".." or ".")
    && !path.Any(c => char.IsControl(c) || "<>\"|?*\".Contains(c));
    if (Kind is not ("Desktop" or "Service")) { yield return new("Select Desktop or
Service.", [nameof(Kind)]); }
    if (!PathValid(ConfigurationPath)) { yield return new("Use an absolute local
Windows configuration path.", [nameof(ConfigurationPath)]); }
    if (Kind == "Desktop" && (!PathValid(ExecutablePath) ||
!ExecutablePath!.EndsWith(".exe", StringComparison.OrdinalIgnoreCase) ||
!string.IsNullOrEmpty(ServiceName)))
    { yield return new("Desktop applications need an executable path and no service
name.", [nameof(ExecutablePath)]); }
    if (Kind == "Service" && (string.IsNullOrEmpty(ServiceName) ||
!string.IsNullOrEmpty(ExecutablePath)))
    { yield return new("Service applications need a service name and no executable
path.", [nameof(ServiceName)]); }
}
}

```

DeploymentSettingsResponse.cs

```

namespace BridgeControl.DTOS.Vendor.DeploymentSettings;

/// <summary>
/// Carries deployment settings data returned by the API.
/// </summary>
public sealed record DeploymentSettingsResponse(DeploymentSettings? Settings, InstallerIdentity?
Installer, bool Frozen);

```

InstallerIdentity.cs

```

namespace BridgeControl.DTOS.Vendor.DeploymentSettings;

/// <summary>
/// Represents installer identity data.
/// </summary>
public sealed record InstallerIdentity(Guid ProductCode, Guid? UpgradeCode, string Version);

```

DTOs/Vendor

EffectiveConfigurationResponse.cs

```

using System.Text.Json;

namespace BridgeControl.DTOS.Vendor;

/// <summary>
/// Carries effective configuration data returned by the API.

```

```
/// </summary>
public sealed record EffectiveConfigurationResponse(Guid RevisionId, Guid InstallationId, Guid
ProductVersionId,
    int RevisionNumber, JsonElement Values, IReadOnlyList<Guid> SourceRevisionIds,
DateTimeOffset CreatedAt);
```

EffectiveConfigurationRevisionDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a generated configuration revision's installation, version and file references,
revision number and creation details.
/// </summary>
public class EffectiveConfigurationRevisionDto
{
    public Guid Id { get; set; }

    public Guid InstallationId { get; set; }

    public Guid ProductId { get; set; }

    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }

    public int RevisionNumber { get; set; }

    public Guid? CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }
}
```

EffectiveConfigurationSourceRevisionDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns the link between an effective configuration revision and one of its source revisions.
/// </summary>
public class EffectiveConfigurationSourceRevisionDto
{
    public Guid EffectiveConfigurationRevisionId { get; set; }

    public Guid ConfigurationRevisionId { get; set; }
}
```

FileArtifactDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns file metadata and checksum without exposing its internal storage location or file
contents.
/// </summary>
```

```

public class FileArtifactDto
{
    public Guid Id { get; set; }

    public string ArtifactType { get; set; } = string.Empty;

    public string FileName { get; set; } = string.Empty;

    public string ContentType { get; set; } = string.Empty;

    public string Sha256 { get; set; } = string.Empty;

    public long SizeBytes { get; set; }

    public DateTimeOffset CreatedAt { get; set; }
}

```

GrantTechnicianCustomerAccessRequest.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Identifies the user and customer for a new delegated access grant; the server records the
/// granting user.
/// </summary>
public class GrantTechnicianCustomerAccessRequest
{
    public required Guid UserId { get; set; }

    public required Guid CustomerId { get; set; }
}

```

InstallationDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns an installation's name, customer, product and current effective configuration
/// reference.
/// </summary>
public class InstallationDto
{
    public Guid Id { get; set; }

    public Guid CustomerId { get; set; }

    public Guid ProductId { get; set; }

    public Guid? CurrentEffectiveConfigurationRevisionId { get; set; }

    public string Name { get; set; } = string.Empty;
}

```


DTOs/Vendor/InstallationStateContracts

DesiredStateResponse.cs

```
namespace BridgeControl.DTOs.Vendor.InstallationStateContracts;

/// <summary>
/// Carries desired state data returned by the API.
/// </summary>
public sealed record DesiredStateResponse(Guid InstallationId, Guid ProductVersionId, Guid EffectiveConfigurationRevisionId, DateTimeOffset UpdatedAt);
```

ReportedStateResponse.cs

```
namespace BridgeControl.DTOs.Vendor.InstallationStateContracts;

/// <summary>
/// Carries reported state data returned by the API.
/// </summary>
public sealed record ReportedStateResponse(Guid InstallationId, Guid? InstalledProductVersionId, Guid? EffectiveConfigurationRevisionId, string? ConfigurationHash, string Status, DateTimeOffset ReportedAt);
```

ReportInstallationStateRequest.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.InstallationStateContracts;

/// <summary>
/// Carries input for the report installation state operation.
/// </summary>
public sealed record ReportInstallationStateRequest(Guid? InstalledProductVersionId, Guid? EffectiveConfigurationRevisionId, [property: MaxLength(64)] string? ConfigurationHash, [property: Required, MaxLength(32)] string Status, DateTimeOffset ReportedAt);
```

SetDesiredStateRequest.cs

```
namespace BridgeControl.DTOs.Vendor.InstallationStateContracts;

/// <summary>
/// Carries input for the set desired state operation.
/// </summary>
public sealed record SetDesiredStateRequest(Guid ProductVersionId, Guid EffectiveConfigurationRevisionId);
```

DTOs/Vendor/LicenseContracts

AssignLicenseRequest.cs

```
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.LicenseContracts;

/// <summary>
/// Carries input for the assign license operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public sealed class AssignLicenseRequest { public Guid LicenseId { get; set; } }
```

CreateLicenseRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.LicenseContracts;

/// <summary>
/// Carries input for the create license operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public sealed class CreateLicenseRequest : IValidatableObject
{
    public Guid CustomerId { get; set; }
    public Guid ProductId { get; set; }
    [Required, MaxLength(200)] public string LicenseReference { get; set; } = "";
    public DateTimeOffset ValidFrom { get; set; }
    public DateTimeOffset? ValidUntil { get; set; }
    [Range(1, int.MaxValue)] public int? MaxInstallations { get; set; }

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (ValidFrom == default) { yield return new("ValidFrom is required.",
[nameof(ValidFrom)]); }
        if (ValidUntil <= ValidFrom) { yield return new("ValidUntil must be later than
ValidFrom.", [nameof(ValidUntil)]); }
    }
}
```

InstallationQuery.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.LicenseContracts;

/// <summary>
/// Contains filter and pagination input for installation searches.
/// </summary>
public sealed class InstallationQuery
{
    public Guid? CustomerId { get; set; }
    public Guid? ProductId { get; set; }
```

```

        [MaxLength(32)] public string? Status { get; set; }
    }

```

LicenseResponse.cs

```

namespace BridgeControl.DTOs.Vendor.LicenseContracts;

/// <summary>
/// Carries license data returned by the API.
/// </summary>
public sealed record LicenseResponse(Guid Id, Guid CustomerId, Guid ProductId, string
LicenseReference,
    string Status, DateTimeOffset ValidFrom, DateTimeOffset? ValidUntil, int?
MaxInstallations);

```

UpdateLicenseRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.LicenseContracts;

/// <summary>
/// Carries input for the update license operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public sealed class UpdateLicenseRequest : IValidatableObject
{
    private string? reference;
    private DateTimeOffset? from, until;
    private int? maximum;
    [MaxLength(200)] public string? LicenseReference { get => reference; set { reference =
value?.Trim(); ReferenceProvided = true; } }
    public DateTimeOffset? ValidFrom { get => from; set { from = value; FromProvided = true; }
}
    public DateTimeOffset? ValidUntil { get => until; set { until = value; UntilProvided =
true; } }
    [Range(1, int.MaxValue)] public int? MaxInstallations { get => maximum; set { maximum =
value; MaximumProvided = true; } }
    [JsonIgnore] public bool ReferenceProvided { get; private set; }
    [JsonIgnore] public bool FromProvided { get; private set; }
    [JsonIgnore] public bool UntilProvided { get; private set; }
    [JsonIgnore] public bool MaximumProvided { get; private set; }

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (ReferenceProvided && string.IsNullOrEmpty(LicenseReference)) { yield
return new("A supplied reference cannot be empty.", [nameof(LicenseReference)]); }
        if (FromProvided && (ValidFrom is null || ValidFrom == default(DateTimeOffset))) {
yield return new("A supplied ValidFrom must be a timestamp.", [nameof(ValidFrom)]); }
    }
}

```

DTOs/Vendor

LoginVendorResponse.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>A public login choice; no tenant identifiers or infrastructure metadata.</summary>
public sealed record LoginVendorResponse(string Name);
```

MigrationRuleDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a migration step's order, operation, paths, transformation and destructive flag.
/// </summary>
public class MigrationRuleDto
{
    public Guid Id { get; set; }

    public Guid MigrationId { get; set; }

    public int Sequence { get; set; }

    public string RuleType { get; set; } = string.Empty;

    public string SourcePath { get; set; } = string.Empty;

    public string TargetPath { get; set; } = string.Empty;

    public string Transform { get; set; } = string.Empty;

    public bool IsDestructive { get; set; }
}
```

MigrationRuleRequest.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Carries input for the migration rule operation.
/// </summary>
public sealed record MigrationRuleRequest(int Sequence, string RuleType, string? SourcePath,
string? TargetPath, string? Transform, bool IsDestructive);
```

DTOs/Vendor/OperationalHistoryContracts

InstallationHealthResponse.cs

```

namespace BridgeControl.DTOs.Vendor.OperationalHistoryContracts;

/// <summary>
/// Carries installation health data returned by the API.
/// </summary>
public sealed record InstallationHealthResponse(Guid InstallationId, string Connectivity, string
Freshness, string Drift,
    string Configuration, string License, string Deployment, bool NeedsAttention);

```

LogResponse.cs

```

namespace BridgeControl.DTOs.Vendor.OperationalHistoryContracts;

/// <summary>
/// Carries log data returned by the API.
/// </summary>
public sealed record LogResponse(long Id, Guid AgentId, Guid? InstallationId, string Severity,
string Source, string Message, DateTimeOffset CreatedAt);

```

LogSearchQuery.cs

```

using System.ComponentModel.DataAnnotations;
namespace BridgeControl.DTOs.Vendor.OperationalHistoryContracts;

/// <summary>
/// Contains filter and pagination input for log search searches.
/// </summary>
public sealed class LogSearchQuery : IValidatableObject
{
    public Guid? AgentId { get; set; }
    public Guid? InstallationId { get; set; }
    [MaxLength(16)] public string? Severity { get; set; }
    public DateTimeOffset? From { get; set; }
    public DateTimeOffset? To { get; set; }

    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (From > To) { yield return new("From must not be later than To."); }
        if (Severity is not null && !LogSeverities.Values.Contains(Severity)) { yield
return new("Unknown severity."); }
    }
}

```

LogSeverities.cs

```

namespace BridgeControl.DTOs.Vendor.OperationalHistoryContracts;

/// <summary>
/// Defines the accepted operational log severity values.
/// </summary>
public static class LogSeverities
{

```

```

        public static readonly string[] Values = ["Trace", "Debug", "Information", "Warning",
"Error", "Critical"];
    }

```

MonitoringOverviewResponse.cs

```

using BridgeControl.DTOs.Shared;
namespace BridgeControl.DTOs.Vendor.OperationalHistoryContracts;

/// <summary>
/// Carries monitoring overview data returned by the API.
/// </summary>
public sealed record MonitoringOverviewResponse(DateTimeOffset GeneratedAt, int
TotalInstallations, int OnlineInstallations,
    int StaleInstallations, int OfflineInstallations, int UnknownInstallations, int
DriftedInstallations,
    int IneligibleInstallations, int FailedDeployments, int PendingDeployments, int
ConfigurationFailures, int AttentionCount,
    PagedResponse<InstallationHealthResponse> Installations);

```

DTOs/Vendor

PermissionDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a permission's identity, code and description for access management.
/// </summary>
public class PermissionDto
{
    public Guid Id { get; set; }

    public string Code { get; set; } = string.Empty;

    public string Description { get; set; } = string.Empty;
}

```

ProductDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a product's identity, name, description, status and creation time.
/// </summary>
public class ProductDto
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string? Description { get; set; }
}

```

```

        public string Status { get; set; } = string.Empty;

        public DateTimeOffset CreatedAt { get; set; }
    }

```

ProductVersionArtifactDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns the association between a product version and a stored file.
/// </summary>
public class ProductVersionArtifactDto
{
    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }
}

```

ProductVersionDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a product release's version label, status and creation and publication dates.
/// </summary>
public class ProductVersionDto
{
    public Guid Id { get; set; }

    public Guid ProductId { get; set; }

    public string Version { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? PublishedAt { get; set; }
}

```

RoleDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a role's identity, name and scope for access management.
/// </summary>
public class RoleDto
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;
}

```

```
        public string Scope { get; set; } = string.Empty;
    }
```

RolePermissionDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns the assignment of a permission to a role.
/// </summary>
public class RolePermissionDto
{
    public Guid Id { get; set; }

    public Guid RoleId { get; set; }

    public Guid PermissionId { get; set; }
}
```

TechnicianCustomerAccessDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a user's customer access grant, including who granted it and when.
/// </summary>
public class TechnicianCustomerAccessDto
{
    public Guid Id { get; set; }

    public Guid UserId { get; set; }

    public Guid CustomerId { get; set; }

    public Guid GrantedByUserId { get; set; }

    public DateTimeOffset GrantedAt { get; set; }
}
```

DTOs/Vendor/TenantAuthenticationDtos

AuthSessionResponse.cs

```
namespace BridgeControl.DTOs.Vendor.TenantAuthenticationDtos;

/// <summary>
/// Carries auth session data returned by the API.
/// </summary>
public class AuthSessionResponse
{
    public Guid UserId { get; set; }
    public Guid VendorId { get; set; }
}
```



```

        public string Email { get; set; } = string.Empty;
        public DateTimeOffset AccessTokenExpiresAt { get; set; }
        public string AccessToken { get; set; } = string.Empty;
        public IReadOnlyList<string> Roles { get; set; } = [];
        public IReadOnlyList<string> Permissions { get; set; } = [];
        public Guid? CustomerId { get; set; }
    }

```

ChangePasswordRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;
using BridgeControl.Api;

namespace BridgeControl.DTOS.Vendor.TenantAuthenticationDtos;

/// <summary>
/// Carries input for the change password operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class ChangePasswordRequest
{
    [Required, PreserveWhitespace, StringLength(128)] public string CurrentPassword { get; set; } = string.Empty;
    [Required, PreserveWhitespace, StringLength(128, MinimumLength = 10)] public string NewPassword { get; set; } = string.Empty;
}

```

ForgotPasswordRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOS.Vendor.TenantAuthenticationDtos;

/// <summary>
/// Carries input for the forgot password operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class ForgotPasswordRequest
{
    [Required, StringLength(200)] public string VendorName { get; set; } = string.Empty;
    [Required, EmailAddress, StringLength(254)] public string Email { get; set; } = string.Empty;
}

```

LoginRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;
using BridgeControl.Api;

namespace BridgeControl.DTOS.Vendor.TenantAuthenticationDtos;

/// <summary>

```

```

/// Carries input for the login operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class LoginRequest
{
    [Required, StringLength(200)] public string VendorName { get; set; } = string.Empty;
    [Required, EmailAddress, StringLength(254)] public string Email { get; set; } =
string.Empty;
    [Required, PreserveWhitespace, StringLength(128)] public string Password { get; set; } =
string.Empty;
}

```

ResetPasswordRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;
using BridgeControl.Api;

namespace BridgeControl.DTOs.Vendor.TenantAuthenticationDtos;

/// <summary>
/// Carries input for the reset password operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class ResetPasswordRequest
{
    [Required, PreserveWhitespace, StringLength(256)] public string Token { get; set; } =
string.Empty;
    [Required, PreserveWhitespace, StringLength(128, MinimumLength = 10)] public string
Password { get; set; } = string.Empty;
}

```

DTOs/Vendor

UpdateConfigurationRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Carries input for the update configuration operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public sealed class UpdateConfigurationRequest : IValidatableObject
{
    public required Guid ProductVersionId { get; set; }
    public required JsonElement Values { get; set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)

```

```

        {
            if (Values.ValueKind != JsonValueKind.Object) { yield return new("Configuration
values must be an object.", [nameof(Values)]); }
        }
    }
}

```

UpdateCustomerRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Changes only a supplied name; lifecycle and ownership are separate operations.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateCustomerRequest : IValidatableObject
{
    private string? name;
    [StringLength(200)] public string? Name { get => name; set { name = value?.Trim();
NameProvided = true; } }
    [JsonIgnore] public bool NameProvided { get; private set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (NameProvided && string.IsNullOrEmpty(Name)) { yield return new("A
supplied name cannot be empty.", [nameof(Name)]); }
    }
}

```

UpdateInstallationRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Supplies a new name for an existing installation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateInstallationRequest : IValidatableObject
{
    private string? name;
    [MaxLength(200)] public string? Name { get => name; set { name = value?.Trim();
NameProvided = true; } }
    [JsonIgnore] public bool NameProvided { get; private set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (NameProvided && string.IsNullOrEmpty(Name)) { yield return new("A
supplied name cannot be empty.", [nameof(Name)]); }
    }
}

```

```

    }
}

```

UpdateProductRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Carries input for the update product operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateProductRequest : IValidatableObject
{
    private string? name;
    private string? description;
    [StringLength(200)] public string? Name { get => name; set { name = value?.Trim();
NameProvided = true; } }
    [StringLength(2000)] public string? Description { get => description; set { description =
value?.Trim(); DescriptionProvided = true; } }
    [JsonIgnore] public bool NameProvided { get; private set; }
    [JsonIgnore] public bool DescriptionProvided { get; private set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (NameProvided && string.IsNullOrEmpty(Name)) { yield return new("A
supplied name cannot be empty.", [nameof(Name)]); }
    }
}

```

UpdateProductVersionRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Carries input for the update product version operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateProductVersionRequest : IValidatableObject
{
    private string? version;
    [StringLength(100)] public string? Version { get => version; set { version =
value?.Trim(); VersionProvided = true; } }
    [JsonIgnore] public bool VersionProvided { get; private set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {

```

```

        if (VersionProvided && string.IsNullOrWhiteSpace(Version)) { yield return new("A
supplied version cannot be empty.", [nameof(Version)]); }
    }
}

```

UserAccountDto.cs

```

namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns a user's account details and optional customer membership without exposing the
password hash.
/// </summary>
public class UserAccountDto
{
    public Guid Id { get; set; }

    public Guid? CustomerId { get; set; }

    public string Email { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? LastLoginAt { get; set; }
}

```

DTOs/Vendor/UserAdministrationRequests

AssignmentValidation.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Validates collections of identifiers used for access assignments.
/// </summary>
internal static class AssignmentValidation
{
    /// <summary>
    /// Reports missing, empty, or duplicate assignment identifiers.
    /// </summary>
    public static IEnumerable<ValidationResult> Distinct(Guid[]? ids, string field)
    {
        if (ids is not null && (ids.Contains(Guid.Empty) || ids.Distinct().Count() !=
ids.Length))
        { yield return new("Identifiers must be nonempty and distinct.", [field]); }
    }
}

```

CreateUserRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries input for the create user operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class CreateUserRequest : IValidatableObject
{
    [Required, EmailAddress, StringLength(254)] public string Email { get; set; } =
string.Empty;
    public Guid? CustomerId { get; set; }
    [Required, MinLength(1), MaxLength(4)] public Guid[] RoleIds { get; set; } = [];
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context) =>
AssignmentValidation.Distinct(RoleIds, nameof(RoleIds));
}
```

CustomerQuery.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Contains filter and pagination input for customer searches.
/// </summary>
public class CustomerQuery
{
    [StringLength(200)] public string? Name { get; set; }
    [StringLength(32)] public string? Status { get; set; }
}
```

CustomerResponse.cs

```
namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries customer data returned by the API.
/// </summary>
public class CustomerResponse
{
    public Guid Id { get; set; }
    public string Name { get; set; } = string.Empty;
    public string Status { get; set; } = string.Empty;
    public DateTimeOffset CreatedAt { get; set; }
}
```

SetCustomerAccessRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries input for the set customer access operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class SetCustomerAccessRequest : IValidatableObject
{
    [Required, MaxLength(1000)] public required Guid[] CustomerIds { get; set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context) =>
AssignmentValidation.Distinct(CustomerIds, nameof(CustomerIds));
}
```

SetInstallationAccessRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries input for the set installation access operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class SetInstallationAccessRequest : IValidatableObject
{
    [Required, MaxLength(1000)] public required Guid[] InstallationIds { get; set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context) =>
AssignmentValidation.Distinct(InstallationIds, nameof(InstallationIds));
}
```

SetUserRolesRequest.cs

```
using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries input for the set user roles operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class SetUserRolesRequest : IValidatableObject
{
    [Required, MaxLength(4)] public required Guid[] RoleIds { get; set; }
    /// <summary>
```

```

        /// Validates supplied fields and reports violations of the request constraints.
        /// </summary>
        public IEnumerable<ValidationResult> Validate(ValidationContext context) =>
AssignmentValidation.Distinct(RoleIds, nameof(RoleIds));
    }

```

UpdateUserRequest.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json.Serialization;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Carries input for the update user operation.
/// </summary>
[JsonUnmappedMemberHandling(JsonUnmappedMemberHandling.Disallow)]
public class UpdateUserRequest : IValidatableObject
{
    private string? email;
    [EmailAddress, StringLength(254)] public string? Email { get => email; set { email =
value?.Trim().ToLowerInvariant(); EmailProvided = true; } }
    [JsonIgnore] public bool EmailProvided { get; private set; }
    /// <summary>
    /// Validates supplied fields and reports violations of the request constraints.
    /// </summary>
    public IEnumerable<ValidationResult> Validate(ValidationContext context)
    {
        if (EmailProvided && string.IsNullOrEmpty(Email)) { yield return new("A
supplied email cannot be empty.", [nameof(Email)]); }
    }
}

```

UserQuery.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.DTOs.Vendor.UserAdministrationRequests;

/// <summary>
/// Contains filter and pagination input for user searches.
/// </summary>
public class UserQuery
{
    public Guid? CustomerId { get; set; }
    [StringLength(254)] public string? Email { get; set; }
    [StringLength(32)] public string? Status { get; set; }
}

```

DTOs/Vendor

UserResponse.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns the documented user identity, role names and lifecycle state without credential data.
/// </summary>
public class UserResponse
{
    public Guid Id { get; set; }
    public Guid? CustomerId { get; set; }
    public string Email { get; set; } = string.Empty;
    public string Status { get; set; } = string.Empty;
    public IReadOnlyList<string> Roles { get; set; } = [];
    public DateTimeOffset CreatedAt { get; set; }
}
```

UserRoleDto.cs

```
namespace BridgeControl.DTOs.Vendor;

/// <summary>
/// Returns the assignment of a role to a user account.
/// </summary>
public class UserRoleDto
{
    public Guid Id { get; set; }

    public Guid UserId { get; set; }

    public Guid RoleId { get; set; }
}
```

Mappings/Shared

EntityMapper.cs

```
using System.Reflection;

namespace BridgeControl.Mappings.Shared;

/// <summary>
/// Copies matching simple properties from a DTO into an existing entity, as in BookPub.
/// </summary>
public static class EntityMapper
{
    /// <summary>
    /// Use a validated create/update DTO containing only editable fields. Matching null and
    default values are copied too.
    /// </summary>
    public static void MapDto<TDto, TEntity>(this TEntity entity, TDto dto)
        where TDto : class
        where TEntity : class
    {

```

```

        ArgumentNullException.ThrowIfNull(entity);
        ArgumentNullException.ThrowIfNull(dto);

        foreach (PropertyInfo dtoProperty in
typeof(TDto).GetProperties(BindingFlags.Public | BindingFlags.Instance))
        {
            PropertyInfo? entityProperty =
typeof(TEntity).GetProperty(dtoProperty.Name, BindingFlags.Public | BindingFlags.Instance);
            if (dtoProperty.GetMethod?.IsPublic == true &&
entityProperty?.SetMethod?.IsPublic == true
                && dtoProperty.GetIndexParameters().Length == 0 &&
entityProperty.GetIndexParameters().Length == 0
                && dtoProperty.PropertyType == entityProperty.PropertyType
                && (dtoProperty.PropertyType.IsValueType ||
dtoProperty.PropertyType == typeof(string)))
            {
                entityProperty.SetValue(entity, dtoProperty.GetValue(dto));
            }
        }
    }
}

```

Models/AgentModels

Agent.cs

```

using System.ComponentModel.DataAnnotations;
using BridgeControl.Models.Vendor;

namespace BridgeControl.Models.AgentModels;

/// <summary>
/// Represents agent data.
/// </summary>
public class Agent
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid CustomerId { get; set; }
    [MaxLength(200)] public string MachineName { get; set; } = "";
    [MaxLength(100)] public string AgentVersion { get; set; } = "";
    [MaxLength(32)] public string Status { get; set; } = "Active";
    [MaxLength(100)] public string? ReportedStatus { get; set; }
    public DateTimeOffset RegisteredAt { get; set; }
    public DateTimeOffset? LastSeenAt { get; set; }
    public DateTimeOffset? LastReportedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
    public Customer Customer { get; set; } = null!;
}

```

AgentCredential.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.AgentModels;

```

```

/// <summary>
/// Represents agent credential data.
/// </summary>
public class AgentCredential
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid AgentId { get; set; }
    [MaxLength(32)] public byte[] SecretHash { get; set; } = [];
    [MaxLength(32)] public string Purpose { get; set; } = "AgentAuthentication";
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset? RevokedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
    public Agent Agent { get; set; } = null!;
}

```

AgentEnrollment.cs

```

using System.ComponentModel.DataAnnotations;
using BridgeControl.Models.Vendor;

namespace BridgeControl.Models.AgentModels;

/// <summary>
/// Represents agent enrollment data.
/// </summary>
public class AgentEnrollment
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid CustomerId { get; set; }
    public Guid CreatedBy { get; set; }
    [MaxLength(32)] public byte[] SecretHash { get; set; } = [];
    [MaxLength(32)] public string Purpose { get; set; } = "AgentEnrollment";
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset? UsedAt { get; set; }
    public DateTimeOffset? RevokedAt { get; set; }
    public Guid? AgentId { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
    public Customer Customer { get; set; } = null!;
}

```

AgentHeartbeat.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.AgentModels;

/// <summary>
/// Represents agent heartbeat data.
/// </summary>
public class AgentHeartbeat
{
    public long Id { get; set; }
    public Guid AgentId { get; set; }
    [MaxLength(100)] public string AgentVersion { get; set; } = "";
}

```

```

        [MaxLength(100)] public string Status { get; set; } = "";
        public DateTimeOffset ReportedAt { get; set; }
        public DateTimeOffset ReceivedAt { get; set; }
    }

```

Models/Platform

PlatformAdmin.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using BridgeControl.Models.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Platform;

/// <summary>
/// Stores a platform administrator's sign-in details and account status, with links to their
/// audit history.
/// </summary>
[Index(nameof(Email), IsUnique = true)]
public class PlatformAdmin
{
    public Guid Id { get; set; }

    public string Email { get; set; } = string.Empty;

    public string PasswordHash { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? LastLoginAt { get; set; }

    public ICollection<AuthenticationSession> Sessions { get; set; } = [];

    [InverseProperty(nameof(PlatformAuditLog.PlatformAdmin))]
    public ICollection<PlatformAuditLog> AuditLogs { get; set; } = [];
}

```

PlatformAuditLog.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Platform;

/// <summary>
/// Records a platform administrator's action, the affected entity and, when applicable, the
/// vendor.
/// </summary>
public class PlatformAuditLog
{
    public Guid Id { get; set; }
}

```

```

    public Guid PlatformAdminId { get; set; }

    public Guid? VendorId { get; set; }

    public string Action { get; set; } = string.Empty;

    public string EntityType { get; set; } = string.Empty;

    public string EntityId { get; set; } = string.Empty;

    public string Details { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(Platform.PlatformAdmin.AuditLogs))]
    [ForeignKey(nameof(PlatformAdminId))]
    public PlatformAdmin PlatformAdmin { get; set; } = null!;

    [InverseProperty(nameof(SoftwareVendor.AuditLogs))]
    [ForeignKey(nameof(VendorId))]
    public SoftwareVendor? Vendor { get; set; }
}

```

SoftwareVendor.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Platform;

/// <summary>
/// Represents a software vendor on the platform and links it to its isolated database and file
/// archive.
/// </summary>
public class SoftwareVendor
{
    private string name = string.Empty;

    public Guid Id { get; set; }

    public string Name
    {
        get => name;
        set => name = value?.Trim() ?? string.Empty;
    }

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? DeactivatedAt { get; set; }

    // Monotonic session cutoff survives reactivation and interrupted cross-database cleanup.
    public DateTimeOffset? SessionsRevokedBefore { get; set; }

    public byte[] RowVersion { get; set; } = [];

    public VendorProvisioning? Provisioning { get; set; }

    [InverseProperty(nameof(VendorDatabase.Vendor))]
    public VendorDatabase? Database { get; set; }
}

```

```

        [InverseProperty(nameof(VendorFileArchive.Vendor))]
        public VendorFileArchive? FileArchive { get; set; }

        [InverseProperty(nameof(PlatformAuditLog.Vendor))]
        public ICollection<PlatformAuditLog> AuditLogs { get; set; } = [];
    }

```

VendorDatabase.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Platform;

/// <summary>
/// Identifies a vendor's database and the secret reference used to connect to it.
/// </summary>
[Index(nameof(VendorId), IsUnique = true)]
public class VendorDatabase
{
    public Guid Id { get; set; }

    public Guid VendorId { get; set; }

    public string DatabaseName { get; set; } = string.Empty;

    public string ConnectionSecretRef { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(SoftwareVendor.Database))]
    [ForeignKey(nameof(VendorId))]
    public SoftwareVendor Vendor { get; set; } = null!;
}

```

VendorFileArchive.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Platform;

/// <summary>
/// Identifies a vendor's file storage and the secret reference used to access it.
/// </summary>
[Index(nameof(VendorId), IsUnique = true)]
public class VendorFileArchive
{
    public Guid Id { get; set; }

    public Guid VendorId { get; set; }

    public string StorageIdentifier { get; set; } = string.Empty;
}

```

```

    public string AccessSecretRef { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(SoftwareVendor.FileArchive))]
    [ForeignKey(nameof(VendorId))]
    public SoftwareVendor Vendor { get; set; } = null!;
}

```

VendorMaintenanceOperation.cs

```

namespace BridgeControl.Models.Platform;

/// <summary>Platform-owned recovery journal; survives replacement of a vendor
database.</summary>
public sealed class VendorMaintenanceOperation
{
    public Guid Id { get; set; }
    public Guid VendorId { get; set; }
    public Guid PlatformAdminId { get; set; }
    public string Kind { get; set; } = "";
    public string Status { get; set; } = "Running";
    public bool IsActive { get; set; } = true;
    public string Stage { get; set; } = "Started";
    public string SourceSchema { get; set; } = "";
    public string CurrentSchema { get; set; } = "";
    public string TargetSchema { get; set; } = "";
    public Guid? BackupId { get; set; }
    public string? BundleSha256 { get; set; }
    public string? FailureCode { get; set; }
    public string OperatorAccount { get; set; } = "";
    public string OperatorHost { get; set; } = "";
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
    public DateTimeOffset? CompletedAt { get; set; }
    public DateTimeOffset? ExpiredAt { get; set; }
    public byte[] RowVersion { get; set; } = [];
}

```

VendorProvisioning.cs

```

namespace BridgeControl.Models.Platform;

/// <summary>
/// Records completed provisioning steps so an interrupted vendor setup can resume safely.
/// </summary>
public class VendorProvisioning
{
    public Guid VendorId { get; set; }
    public DateTimeOffset? DatabaseReadyAt { get; set; }
    public DateTimeOffset? ArchiveReadyAt { get; set; }
    public DateTimeOffset? SeededAt { get; set; }
    public DateTimeOffset? InvitationEstablishedAt { get; set; }
    public string SchemaVersion { get; set; } = string.Empty;
    public string? FailureCode { get; set; }
}

```

```

        public DateTimeOffset UpdatedAt { get; set; }
        public SoftwareVendor Vendor { get; set; } = null!;
    }

```

Models/Shared

AuthenticationSession.cs

```

namespace BridgeControl.Models.Shared;

/// <summary>
/// Tracks an account's login session, its expiry and revocation in the account's own database.
/// </summary>
public class AuthenticationSession
{
    public Guid Id { get; set; }

    // References PlatformAdmin in the platform database or UserAccount in a vendor database.
    public Guid AccountId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset ExpiresAt { get; set; }

    public DateTimeOffset? LastRefreshedAt { get; set; }

    public DateTimeOffset? RevokedAt { get; set; }

    public byte[] RowVersion { get; set; } = [];

    public ICollection<RefreshToken> RefreshTokens { get; set; } = [];
}

```

RefreshToken.cs

```

namespace BridgeControl.Models.Shared;

/// <summary>
/// Stores a refresh credential's hash, expiry and use history so its session can detect reuse.
/// </summary>
public class RefreshToken
{
    public Guid Id { get; set; }

    public Guid AuthenticationSessionId { get; set; }

    // SHA-256 hash of a random refresh credential; never store the credential itself.
    public byte[] TokenHash { get; set; } = [];

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset ExpiresAt { get; set; }

    public DateTimeOffset? UsedAt { get; set; }
}

```



```

        public byte[] RowVersion { get; set; } = [];

        public AuthenticationSession AuthenticationSession { get; set; } = null!;
    }

```

Models/Vendor

AccountActivation.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Stores a single-use activation hash and encrypted pending email data for an invited user.
/// </summary>
public class AccountActivation
{
    public Guid Id { get; set; }
    public Guid UserId { get; set; }
    public byte[] TokenHash { get; set; } = [];
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset? UsedAt { get; set; }
    public DateTimeOffset? RevokedAt { get; set; }
    public DateTimeOffset? DeliveredAt { get; set; }
    public string? ProtectedToken { get; set; }
    public byte[] RowVersion { get; set; } = [];
    public UserAccount User { get; set; } = null!;
}

```

ConfigurationMigration.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Defines a configuration migration between two product versions and groups its ordered
/// transformation rules.
/// </summary>
public class ConfigurationMigration
{
    public Guid Id { get; set; }

    public Guid FromProductVersionId { get; set; }

    public Guid ToProductVersionId { get; set; }

    public Guid CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    public int RevisionNumber { get; set; }
}

```

```

[System.ComponentModel.DataAnnotations.Timestamp]
public byte[] RowVersion { get; set; } = [];

[InverseProperty(nameof(ProductVersion.OutgoingMigrations))]
[ForeignKey(nameof(FromProductVersionId))]
public ProductVersion FromProductVersion { get; set; } = null!;

[InverseProperty(nameof(ProductVersion.IncomingMigrations))]
[ForeignKey(nameof(ToProductVersionId))]
public ProductVersion ToProductVersion { get; set; } = null!;

[InverseProperty(nameof(UserAccount.ConfigurationMigrations))]
[ForeignKey(nameof(CreatedByUserId))]
public UserAccount CreatedByUser { get; set; } = null!;

[InverseProperty(nameof(MigrationRule.Migration))]
public ICollection<MigrationRule> Rules { get; set; } = [];
}

```

ConfigurationMigrationReview.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>Server-side review snapshot and atomic execution receipt. Never used as an
authorization grant.</summary>
public sealed class ConfigurationMigrationReview
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid UserId { get; set; }
    public Guid InstallationId { get; set; }
    public Guid TargetProductVersionId { get; set; }
    [MaxLength(64)] public string Fingerprint { get; set; } = "";
    public string PlanJson { get; set; } = "{}";
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset? AppliedAt { get; set; }
    public Guid? ResultRevisionId { get; set; }
    public string? ApprovedRuleIdsJson { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
}

```

ConfigurationProfile.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Groups revisions for product defaults, customer overrides for a product, or installation
overrides.
/// </summary>
public class ConfigurationProfile
{
    public Guid Id { get; set; }
    public int LastRevisionNumber { get; set; }
    public byte[] RowVersion { get; set; } = [];
}

```

```

    public Guid? ProductId { get; set; }

    public Guid? CustomerId { get; set; }

    public Guid? InstallationId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(Vendor.Product.ConfigurationProfiles))]
    [ForeignKey(nameof(ProductId))]
    public Product? Product { get; set; }

    [InverseProperty(nameof(Vendor.Customer.ConfigurationProfiles))]
    [ForeignKey(nameof(CustomerId))]
    public Customer? Customer { get; set; }

    [InverseProperty(nameof(Vendor.Installation.ConfigurationProfiles))]
    [ForeignKey(nameof(InstallationId))]
    public Installation? Installation { get; set; }

    [InverseProperty(nameof(ConfigurationRevision.ConfigurationProfile))]
    public ICollection<ConfigurationRevision> Revisions { get; set; } = [];
}

```

ConfigurationRevision.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Records a numbered configuration revision, its stored file, target product version and
/// author.
/// </summary>
public class ConfigurationRevision
{
    public Guid Id { get; set; }

    public Guid ConfigurationProfileId { get; set; }

    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }
    public Guid? SchemaArtifactId { get; set; }
    public string ValidationStatus { get; set; } = "LegacyUnverified";

    public int RevisionNumber { get; set; }

    public Guid CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(Vendor.ConfigurationProfile.Revisions))]
    [ForeignKey(nameof(ConfigurationProfileId))]
    public ConfigurationProfile ConfigurationProfile { get; set; } = null!;

    [InverseProperty(nameof(Vendor.ProductVersion.ConfigurationRevisions))]
    [ForeignKey(nameof(ProductVersionId))]
    public ProductVersion ProductVersion { get; set; } = null!;
}

```

```

[InverseProperty(nameof(Vendor.FileArtifact.ConfigurationRevisions))]
[ForeignKey(nameof(FileArtifactId))]
public FileArtifact FileArtifact { get; set; } = null!;

[InverseProperty(nameof(UserAccount.ConfigurationRevisions))]
[ForeignKey(nameof(CreatedByUserId))]
public UserAccount CreatedByUser { get; set; } = null!;

[InverseProperty(nameof(EffectiveConfigurationSourceRevision.ConfigurationRevision))]
public ICollection<EffectiveConfigurationSourceRevision> EffectiveConfigurationSources {
get; set; } = [];
}

```

ConfigurationSourceChange.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>Durable source notification and fanout cursor for effective-configuration
work.</summary>
public sealed class ConfigurationSourceChange
{
    public Guid Id { get; set; }
    public Guid ConfigurationRevisionId { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset? ProcessedAt { get; set; }
    public Guid? LastInstallationId { get; set; }
}

```

Customer.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents a vendor's customer and groups its users, installations, configuration profiles
and technician access grants.
/// </summary>
public class Customer
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? DeactivatedAt { get; set; }
    public byte[] RowVersion { get; set; } = [];

    [InverseProperty(nameof(UserAccount.Customer))]
    public ICollection<UserAccount> Users { get; set; } = [];

    [InverseProperty(nameof(TechnicianCustomerAccess.Customer))]
    public ICollection<TechnicianCustomerAccess> TechnicianAccessGrants { get; set; } = [];
}

```

```

    [InverseProperty(nameof(ConfigurationProfile.Customer))]
    public ICollection<ConfigurationProfile> ConfigurationProfiles { get; set; } = [];

    [InverseProperty(nameof(Installation.Customer))]
    public ICollection<Installation> Installations { get; set; } = [];
}

```

CustomerUserInstallationAccess.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Grants a customer user access to an installation belonging to the same customer.
/// </summary>
public class CustomerUserInstallationAccess
{
    public Guid Id { get; set; }
    public Guid CustomerId { get; set; }
    public Guid UserId { get; set; }
    public Guid InstallationId { get; set; }
    public Guid GrantedByUserId { get; set; }
    public DateTimeOffset GrantedAt { get; set; }
    public UserAccount User { get; set; } = null!;
    public Installation Installation { get; set; } = null!;
    public UserAccount GrantedByUser { get; set; } = null!;
}

```

Deployment.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents deployment data.
/// </summary>
public sealed class Deployment
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid InstallationId { get; set; }
    public Guid DesiredStateRevisionId { get; set; }
    public Guid TargetProductVersionId { get; set; }
    public Guid TargetConfigurationRevisionId { get; set; }
    public long ReconciliationGeneration { get; set; }
    [MaxLength(32)] public string OperationType { get; set; } = "Install";
    [MaxLength(32)] public string Status { get; set; } = "Pending";
    public bool IsActive { get; set; } = true;
    public Guid? ReconciledReportId { get; set; }
    public DateTimeOffset RequestedAt { get; set; }
    public DateTimeOffset? CompletedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
}

```

DeploymentAttempt.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents deployment attempt data.
/// </summary>
public sealed class DeploymentAttempt
{
    public Guid InstallationId { get; set; }
    public Guid AgentId { get; set; }
    [MaxLength(64)] public string? LastReportHash { get; set; }
    public DateTimeOffset? ResultReceivedAt { get; set; }
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid DeploymentId { get; set; }
    public int AttemptNumber { get; set; }
    [MaxLength(32)] public string Status { get; set; } = "Running";
    [MaxLength(100)] public string? ErrorCode { get; set; }
    [MaxLength(2000)] public string? ErrorMessage { get; set; }
    public DateTimeOffset StartedAt { get; set; }
    public DateTimeOffset CompletedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
}

```

DesiredStateRevision.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents desired state revision data.
/// </summary>
public sealed class DesiredStateRevision
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid InstallationId { get; set; }
    public Guid ProductVersionId { get; set; }
    public Guid EffectiveConfigurationRevisionId { get; set; }
    public long RevisionNumber { get; set; }
    public Guid CreatedByUserId { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
}

```

EffectiveConfigurationRevision.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Records a generated configuration for an installation, including its product version, stored
/// file and source revisions.
/// </summary>
public class EffectiveConfigurationRevision
{
    public Guid Id { get; set; }
}

```

```

    public Guid InstallationId { get; set; }

    /// <summary>
    /// Must match both the installation's product and the product version's product.
    /// </summary>
    public Guid ProductId { get; set; }

    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }
    public Guid? SchemaArtifactId { get; set; }
    public string? InputFingerprint { get; set; }

    public int RevisionNumber { get; set; }

    public Guid? CreatedByUserId { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(Vendor.Installation.EffectiveConfigurationRevisions))]
    [ForeignKey(nameof(InstallationId) + "," + nameof(ProductId))]
    public Installation Installation { get; set; } = null!;

    [InverseProperty(nameof(Vendor.ProductVersion.EffectiveConfigurationRevisions))]
    [ForeignKey(nameof(ProductVersionId) + "," + nameof(ProductId))]
    public ProductVersion ProductVersion { get; set; } = null!;

    [InverseProperty(nameof(Vendor.FileArtifact.EffectiveConfigurationRevisions))]
    [ForeignKey(nameof(FileArtifactId))]
    public FileArtifact FileArtifact { get; set; } = null!;

    [InverseProperty(nameof(UserAccount.EffectiveConfigurationRevisions))]
    [ForeignKey(nameof(CreatedByUserId))]
    public UserAccount? CreatedByUser { get; set; }

    [InverseProperty(nameof(EffectiveConfigurationSourceRevision.EffectiveConfigurationRevision))]
    public ICollection<EffectiveConfigurationSourceRevision> SourceRevisions { get; set; } =
    [];

    [InverseProperty(nameof(Vendor.Installation.CurrentEffectiveConfigurationRevision))]
    public ICollection<Installation> CurrentForInstallations { get; set; } = [];
}

```

EffectiveConfigurationSignal.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>Durable wakeup for a schema/publication change or a newly registered
installation.</summary>
public sealed class EffectiveConfigurationSignal
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid? ProductVersionId { get; set; }
    public Guid? InstallationId { get; set; }
    public Guid? LastTargetId { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset? ProcessedAt { get; set; }
}

```

EffectiveConfigurationSourceRevision.cs

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Links an effective configuration to a source revision so its composition can be traced.
/// </summary>
[PrimaryKey(nameof(EffectiveConfigurationRevisionId), nameof(ConfigurationRevisionId))]
public class EffectiveConfigurationSourceRevision
{
    public Guid EffectiveConfigurationRevisionId { get; set; }

    public Guid ConfigurationRevisionId { get; set; }

    [InverseProperty(nameof(Vendor.EffectiveConfigurationRevision.SourceRevisions))]
    [ForeignKey(nameof(EffectiveConfigurationRevisionId))]
    public EffectiveConfigurationRevision EffectiveConfigurationRevision { get; set; } =
    null!;

    [InverseProperty(nameof(Vendor.ConfigurationRevision.EffectiveConfigurationSources))]
    [ForeignKey(nameof(ConfigurationRevisionId))]
    public ConfigurationRevision ConfigurationRevision { get; set; } = null!;
}
```

EffectiveConfigurationWork.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents effective configuration work data.
/// </summary>
public sealed class EffectiveConfigurationWork
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid InstallationId { get; set; }
    public Guid ProductVersionId { get; set; }
    public long RequestedGeneration { get; set; } = 1;
    public long CompletedGeneration { get; set; }
    [MaxLength(32)] public string Status { get; set; } = "Pending";
    public int Attempts { get; set; }
    public DateTimeOffset NextAttemptAt { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
    [MaxLength(2000)] public string? LastError { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
}
```

FileArtifact.cs

```
using System.ComponentModel.DataAnnotations.Schema;
```



```

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Stores file metadata, storage location and checksum for an installer, schema or configuration
file.
/// </summary>
public class FileArtifact
{
    public string? InstallerIdentityJson { get; set; }
    public Guid Id { get; set; }

    public string ArtifactType { get; set; } = string.Empty;

    public string FileName { get; set; } = string.Empty;

    public string StorageIdentifier { get; set; } = string.Empty;

    public string ContentType { get; set; } = string.Empty;

    public string Sha256 { get; set; } = string.Empty;

    public long SizeBytes { get; set; }

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(ProductVersionArtifact.FileArtifact))]
    public ICollection<ProductVersionArtifact> ProductVersions { get; set; } = [];

    [InverseProperty(nameof(ConfigurationRevision.FileArtifact))]
    public ICollection<ConfigurationRevision> ConfigurationRevisions { get; set; } = [];

    [InverseProperty(nameof(EffectiveConfigurationRevision.FileArtifact))]
    public ICollection<EffectiveConfigurationRevision> EffectiveConfigurationRevisions { get;
set; } = [];
}

```

Installation.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents a customer's installation of a product and tracks its current effective
configuration and configuration history.
/// </summary>
public class Installation
{
    public Guid? DesiredStateRevisionId { get; set; }
    public Guid? ReportedStateReportId { get; set; }
    public long LastDesiredStateRevisionNumber { get; set; }
    // Create the ID before EF tracks the circular current-configuration relationship.
    public Guid Id { get; set; } = Guid.NewGuid();

    public Guid CustomerId { get; set; }

    public Guid ProductId { get; set; }
    public Guid? AgentId { get; set; }
    public string Status { get; set; } = "Active";
}

```

```

    public string? InstalledVersion { get; set; }
    public DateTimeOffset RegisteredAt { get; set; }
    public DateTimeOffset? LastSeenAt { get; set; }

    public Guid? CurrentEffectiveConfigurationRevisionId { get; set; }
    public int LastEffectiveRevisionNumber { get; set; }

    public string Name { get; set; } = string.Empty;
    // Agent retry identity is independent of the human-editable display name.
    public string? RegistrationName { get; set; }
    [System.ComponentModel.DataAnnotations.Timestamp] public byte[] RowVersion { get; set; } =
[];

    [InverseProperty(nameof(ConfigurationProfile.Installation))]
    public ICollection<ConfigurationProfile> ConfigurationProfiles { get; set; } = [];

    [InverseProperty(nameof(EffectiveConfigurationRevision.Installation))]
    public ICollection<EffectiveConfigurationRevision> EffectiveConfigurationRevisions { get;
set; } = [];

    [InverseProperty(nameof(Vendor.Customer.Installations))]
    [ForeignKey(nameof(CustomerId))]
    public Customer Customer { get; set; } = null!;

    [InverseProperty(nameof(Vendor.Product.Installations))]
    [ForeignKey(nameof(ProductId))]
    public Product Product { get; set; } = null!;

    [InverseProperty(nameof(EffectiveConfigurationRevision.CurrentForInstallations))]
    [ForeignKey(nameof(CurrentEffectiveConfigurationRevisionId) + "," + nameof(Id))]
    public EffectiveConfigurationRevision? CurrentEffectiveConfigurationRevision { get; set; }
}

```

InstallationLicenseAssignment.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents installation license assignment data.
/// </summary>
public class InstallationLicenseAssignment
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid InstallationId { get; set; }
    public Guid LicenseId { get; set; }
    public Guid CustomerId { get; set; }
    public Guid ProductId { get; set; }
    public Guid AssignedBy { get; set; }
    public DateTimeOffset AssignedAt { get; set; }
    public Guid? RemovedBy { get; set; }
    public DateTimeOffset? RemovedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
    public Installation Installation { get; set; } = null!;
    public License License { get; set; } = null!;
}

```

InstallationStateReport.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents installation state report data.
/// </summary>
public sealed class InstallationStateReport
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid InstallationId { get; set; }
    public Guid AgentId { get; set; }
    public Guid? InstalledProductVersionId { get; set; }
    public Guid? EffectiveConfigurationRevisionId { get; set; }
    [MaxLength(64)] public string? ConfigurationHash { get; set; }
    [MaxLength(32)] public string Status { get; set; } = "Unknown";
    public DateTimeOffset ReportedAt { get; set; }
    public DateTimeOffset ReceivedAt { get; set; }
}
```

License.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents license data.
/// </summary>
public class License
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid CustomerId { get; set; }
    public Guid ProductId { get; set; }
    [MaxLength(200)] public string LicenseReference { get; set; } = "";
    [MaxLength(32)] public string Status { get; set; } = "Active";
    public DateTimeOffset ValidFrom { get; set; }
    public DateTimeOffset? ValidUntil { get; set; }
    public int? MaxInstallations { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
    public Customer Customer { get; set; } = null!;
    public Product Product { get; set; } = null!;
}
```

LogIngestionWindow.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents log ingestion window data.
/// </summary>
public sealed class LogIngestionWindow
{
    [Key] public Guid AgentId { get; set; }
```

```

        public DateTimeOffset StartedAt { get; set; }
        public int Batches { get; set; }
        public int Entries { get; set; }
    }

```

MigrationDefinitionRevision.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>Immutable rule snapshot, including public rule IDs, retained after definition
replacement.</summary>
public sealed class MigrationDefinitionRevision
{
    public Guid Id { get; set; } = Guid.NewGuid();
    public Guid MigrationId { get; set; }
    public int RevisionNumber { get; set; }
    public string RulesJson { get; set; } = "[]";
    public Guid CreatedByUserId { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
}

```

MigrationRule.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Describes one ordered configuration transformation, including its paths, operation and
whether it is destructive.
/// </summary>
public class MigrationRule
{
    public Guid Id { get; set; }

    public Guid MigrationId { get; set; }

    public int Sequence { get; set; }

    public string RuleType { get; set; } = string.Empty;

    public string SourcePath { get; set; } = string.Empty;

    public string TargetPath { get; set; } = string.Empty;

    public string Transform { get; set; } = string.Empty;

    public bool IsDestructive { get; set; }

    [InverseProperty(nameof(ConfigurationMigration.Rules))]
    [ForeignKey(nameof(MigrationId))]
    public ConfigurationMigration Migration { get; set; } = null!;
}

```

OperationalLog.cs

```

using System.ComponentModel.DataAnnotations;
namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents operational log data.
/// </summary>
public sealed class OperationalLog
{
    public long Id { get; set; }
    public Guid AgentId { get; set; }
    public Guid? InstallationId { get; set; }
    [MaxLength(16)] public string Severity { get; set; } = "Information";
    [MaxLength(100)] public string Source { get; set; } = "";
    [MaxLength(2000)] public string Message { get; set; } = "";
    public bool IsSensitive { get; set; }
    [MaxLength(64)] public string Fingerprint { get; set; } = "";
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ReceivedAt { get; set; }
}

```

PasswordReset.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>Purpose-specific reset verification and encrypted delivery retry state.</summary>
public class PasswordReset
{
    public Guid Id { get; set; }
    public Guid UserId { get; set; }
    public byte[] TokenHash { get; set; } = [];
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset? UsedAt { get; set; }
    public DateTimeOffset? RevokedAt { get; set; }
    public DateTimeOffset? DeliveredAt { get; set; }
    public string? ProtectedToken { get; set; }
    public byte[] RowVersion { get; set; } = [];
    public UserAccount User { get; set; } = null!;
}

```

Permission.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Defines a named action that roles can grant to users.
/// </summary>
[Index(nameof(Code), IsUnique = true)]
public class Permission
{
    public Guid Id { get; set; }

    public string Code { get; set; } = string.Empty;
}

```

```

        public string Description { get; set; } = string.Empty;

        [InverseProperty(nameof(RolePermission.Permission))]
        public ICollection<RolePermission> RolePermissions { get; set; } = [];
    }

```

Product.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents software managed by a vendor and groups its versions, installations and
/// configuration profiles.
/// </summary>
public class Product
{
    public string? DeploymentSettingsJson { get; set; }
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string? Description { get; set; }

    public byte[] RowVersion { get; set; } = [];

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    [InverseProperty(nameof(ProductVersion.Product))]
    public ICollection<ProductVersion> Versions { get; set; } = [];

    [InverseProperty(nameof(ConfigurationProfile.Product))]
    public ICollection<ConfigurationProfile> ConfigurationProfiles { get; set; } = [];

    [InverseProperty(nameof(Installation.Product))]
    public ICollection<Installation> Installations { get; set; } = [];
}

```

ProductVersion.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents a specific product release and links its files, configurations and available
/// configuration migrations.
/// </summary>
public class ProductVersion
{
    public string? DeploymentSettingsJson { get; set; }
    public Guid Id { get; set; }

    public Guid ProductId { get; set; }
}

```

```

public byte[] RowVersion { get; set; } = [];

public string Version { get; set; } = string.Empty;

public string Status { get; set; } = string.Empty;

public DateTimeOffset CreatedAt { get; set; }

public DateTimeOffset? PublishedAt { get; set; }
public Guid? PublishedSchemaArtifactId { get; set; }
public Guid? PublishedInstallerArtifactId { get; set; }
public Guid? PublishedDefaultRevisionId { get; set; }

[InverseProperty(nameof(Vendor.Product.Versions))]
[ForeignKey(nameof(ProductId))]
public Product Product { get; set; } = null!;

[InverseProperty(nameof(ProductVersionArtifact.ProductVersion))]
public ICollection<ProductVersionArtifact> Artifacts { get; set; } = [];

[InverseProperty(nameof(ConfigurationRevision.ProductVersion))]
public ICollection<ConfigurationRevision> ConfigurationRevisions { get; set; } = [];

[InverseProperty(nameof(EffectiveConfigurationRevision.ProductVersion))]
public ICollection<EffectiveConfigurationRevision> EffectiveConfigurationRevisions { get;
set; } = [];

[InverseProperty(nameof(ConfigurationMigration.FromProductVersion))]
public ICollection<ConfigurationMigration> OutgoingMigrations { get; set; } = [];

[InverseProperty(nameof(ConfigurationMigration.ToProductVersion))]
public ICollection<ConfigurationMigration> IncomingMigrations { get; set; } = [];
}

```

ProductVersionArtifact.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Associates a stored file with a product version, allowing a release to reference multiple
artifacts.
/// </summary>
[PrimaryKey(nameof(ProductVersionId), nameof(FileArtifactId))]
public class ProductVersionArtifact
{
    public string ArtifactType { get; set; } = string.Empty;
    public Guid ProductVersionId { get; set; }

    public Guid FileArtifactId { get; set; }

    [InverseProperty(nameof(Vendor.ProductVersion.Artifacts))]
    [ForeignKey(nameof(ProductVersionId))]
    public ProductVersion ProductVersion { get; set; } = null!;

    [InverseProperty(nameof(Vendor.FileArtifact.ProductVersions))]
    [ForeignKey(nameof(FileArtifactId))]

```

```

        public FileArtifact FileArtifact { get; set; } = null!;
    }

```

Role.cs

```

using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Defines a named access role and its scope, with links to assigned users and permissions.
/// </summary>
[Index(nameof(Name), IsUnique = true)]
public class Role
{
    public Guid Id { get; set; }

    public string Name { get; set; } = string.Empty;

    public string Scope { get; set; } = string.Empty;

    [InverseProperty(nameof(UserRole.Role))]
    public ICollection<UserRole> UserRoles { get; set; } = [];

    [InverseProperty(nameof(RolePermission.Role))]
    public ICollection<RolePermission> RolePermissions { get; set; } = [];
}

```

RolePermission.cs

```

using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Links a permission to a role to describe the actions that role grants.
/// </summary>
public class RolePermission
{
    public Guid Id { get; set; }

    public Guid RoleId { get; set; }

    public Guid PermissionId { get; set; }

    [InverseProperty(nameof(Vendor.Role.RolePermissions))]
    [ForeignKey(nameof(RoleId))]
    public Role Role { get; set; } = null!;

    [InverseProperty(nameof(Vendor.Permission.RolePermissions))]
    [ForeignKey(nameof(PermissionId))]
    public Permission Permission { get; set; } = null!;
}

```


StateReconciliationWork.cs

```
using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Represents state reconciliation work data.
/// </summary>
public sealed class StateReconciliationWork
{
    [Key] public Guid InstallationId { get; set; }
    public long RequestedGeneration { get; set; }
    public long CompletedGeneration { get; set; }
    [MaxLength(32)] public string Status { get; set; } = "Pending";
    [MaxLength(100)] public string? Reason { get; set; }
    public int Attempts { get; set; }
    public DateTimeOffset NextAttemptAt { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
    [Timestamp] public byte[] RowVersion { get; set; } = [];
}
```

TechnicianCustomerAccess.cs

```
using System.ComponentModel.DataAnnotations.Schema;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Records a user's delegated access to a customer, including who granted it and when.
/// </summary>
public class TechnicianCustomerAccess
{
    public Guid Id { get; set; }

    public Guid UserId { get; set; }

    public Guid CustomerId { get; set; }

    public Guid GrantedByUserId { get; set; }

    public DateTimeOffset GrantedAt { get; set; }

    [InverseProperty(nameof(UserAccount.CustomerAccessGrants))]
    [ForeignKey(nameof(UserId))]
    public UserAccount User { get; set; } = null!;

    [InverseProperty(nameof(Vendor.Customer.TechnicianAccessGrants))]
    [ForeignKey(nameof(CustomerId))]
    public Customer Customer { get; set; } = null!;

    [InverseProperty(nameof(UserAccount.GrantedCustomerAccess))]
    [ForeignKey(nameof(GrantedByUserId))]
    public UserAccount GrantedByUser { get; set; } = null!;
}
```

UserAccount.cs

```
using System.ComponentModel.DataAnnotations.Schema;
using BridgeControl.Models.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Stores a vendor user account, its credentials and status, with optional customer membership
/// and access relationships.
/// </summary>
[Index(nameof(Email), IsUnique = true)]
public class UserAccount
{
    public Guid Id { get; set; }

    public Guid? CustomerId { get; set; }

    public string Email { get; set; } = string.Empty;

    public string PasswordHash { get; set; } = string.Empty;

    public string Status { get; set; } = string.Empty;

    public DateTimeOffset CreatedAt { get; set; }

    public DateTimeOffset? LastLoginAt { get; set; }
    public byte[] RowVersion { get; set; } = [];

    public ICollection<AuthenticationSession> Sessions { get; set; } = [];

    [InverseProperty(nameof(Vendor.Customer.Users))]
    [ForeignKey(nameof(CustomerId))]
    public Customer? Customer { get; set; }

    [InverseProperty(nameof(UserRole.User))]
    public ICollection<UserRole> UserRoles { get; set; } = [];

    [InverseProperty(nameof(TechnicianCustomerAccess.User))]
    public ICollection<TechnicianCustomerAccess> CustomerAccessGrants { get; set; } = [];

    [InverseProperty(nameof(TechnicianCustomerAccess.GrantedByUser))]
    public ICollection<TechnicianCustomerAccess> GrantedCustomerAccess { get; set; } = [];

    [InverseProperty(nameof(ConfigurationRevision.CreatedByUser))]
    public ICollection<ConfigurationRevision> ConfigurationRevisions { get; set; } = [];

    [InverseProperty(nameof(EffectiveConfigurationRevision.CreatedByUser))]
    public ICollection<EffectiveConfigurationRevision> EffectiveConfigurationRevisions { get;
set; } = [];

    [InverseProperty(nameof(ConfigurationMigration.CreatedByUser))]
    public ICollection<ConfigurationMigration> ConfigurationMigrations { get; set; } = [];
}
```

UserRole.cs

```
using System.ComponentModel.DataAnnotations.Schema;
```

```

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Assigns a role to a user account.
/// </summary>
public class UserRole
{
    public Guid Id { get; set; }

    public Guid UserId { get; set; }

    public Guid RoleId { get; set; }

    [InverseProperty(nameof(UserAccount.UserRoles))]
    [ForeignKey(nameof(UserId))]
    public UserAccount User { get; set; } = null!;

    [InverseProperty(nameof(Role.UserRoles))]
    [ForeignKey(nameof(RoleId))]
    public Role Role { get; set; } = null!;
}

```

VendorAuditLog.cs

```

namespace BridgeControl.Models.Vendor;

/// <summary>
/// Preserves attributable vendor security events without storing credentials or cross-database
foreign keys.
/// </summary>
public class VendorAuditLog
{
    public Guid Id { get; set; }
    public Guid ActorId { get; set; }
    public string ActorType { get; set; } = string.Empty;
    public string Action { get; set; } = string.Empty;
    public string EntityType { get; set; } = "UserAccount";
    public string Outcome { get; set; } = "Succeeded";
    public string Context { get; set; } = "{}";
    public Guid EntityId { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
}

```

Options

AgentAuthenticationOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for agent authentication.

```

```

/// </summary>
public sealed class AgentAuthenticationOptions
{
    public const string SectionName = "Authentication:Agent";
    [Required] public string Issuer { get; set; } = "BridgeControl";
    [Required] public string Audience { get; set; } = "BridgeControl.Agent";
    public string SigningKey { get; set; } = "";
    /// <summary>
    /// Checks that the configured token signing key meets the required length.
    /// </summary>
    public bool HasValidSigningKey()
    {
        try { return Convert.FromBase64String(SigningKey).Length >= 32; }
        catch (FormatException) { return false; }
    }
    /// <summary>
    /// Checks that the agent signing key differs from the other identity's signing key.
    /// </summary>
    public bool HasDistinctSigningKey(string other)
    {
        try { return
!Convert.FromBase64String(SigningKey).SequenceEqual(Convert.FromBase64String(other)); }
        catch (FormatException) { return false; }
    }
}

```

ArtifactArchiveOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for artifact archive.
/// </summary>
public class ArtifactArchiveOptions
{
    public const long MaximumInstallerBytes = 256L * 1024 * 1024;
    [Range(1, MaximumInstallerBytes)] public long InstallerMaxBytes { get; set; } =
MaximumInstallerBytes;
    [Range(1, 1024 * 1024)] public long SchemaMaxBytes { get; set; } = 1024 * 1024;
    [Range(1, 300)] public int UploadTimeoutSeconds { get; set; } = 120;
}

```

DatabaseOperationsOptions.cs

```

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for database operations.
/// </summary>
public sealed class DatabaseOperationsOptions
{
    public string BackupRoot { get; set; } = "";
    public string ActiveKeyId { get; set; } = "";
    public string? SqlServiceSid { get; set; }
    public Dictionary<string, string> EncryptionKeys { get; set; } = [];
}

```

```

        public int RetentionDays { get; set; } = 30;
        public long MaximumBundleBytes { get; set; } = 20L * 1024 * 1024 * 1024;
        public int MaximumArtifacts { get; set; } = 100000;
        public int SqlTimeoutSeconds { get; set; } = 600;
    }

```

EffectiveConfigurationOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for effective configuration.
/// </summary>
public sealed class EffectiveConfigurationOptions
{
    public bool WorkerEnabled { get; set; } = true;
    [Range(1, 300)] public int PollSeconds { get; set; } = 5;
}

```

HostingProfileOptions.cs

```

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for hosting profile.
/// </summary>
public sealed class HostingProfileOptions
{
    public string PublicOrigin { get; set; } = "";
    public bool RequireProxy { get; set; }
    public string[] TrustedProxies { get; set; } = [];
    public string DataProtectionKeyDirectory { get; set; } = "";
    public string LogDirectory { get; set; } =
        Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
            "BridgeControl", "Logs");
}

```

OperationalHistoryOptions.cs

```

using System.ComponentModel.DataAnnotations;
namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for operational history.
/// </summary>
public sealed class OperationalHistoryOptions
{
    [Range(1, 365)] public int LogDays { get; set; } = 30;
    [Range(1, 365)] public int HeartbeatDays { get; set; } = 30;
    [Range(30, 365)] public int CredentialGraceDays { get; set; } = 30;
    [Range(100, 1000000)] public int MaxLogsPerVendor { get; set; } = 100000;
    [Range(1, 1000)] public int CleanupBatchSize { get; set; } = 500;
    public bool WorkerEnabled { get; set; } = true;
}

```

```

        [Range(1, 3600)] public int PollSeconds { get; set; } = 60;
    }

```

PlatformAuthenticationOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Configures platform access-token signing and the lifetime of human sessions.
/// </summary>
public class PlatformAuthenticationOptions
{
    public const string SectionName = "Authentication:Platform";

    [Required]
    public string Issuer { get; set; } = "BridgeControl";

    [Required]
    public string Audience { get; set; } = "BridgeControl.Platform";

    public string SigningKey { get; set; } = string.Empty;

    [Range(1, 30)]
    public int AccessTokenMinutes { get; set; } = 10;

    [Range(1, 30)]
    public int SessionDays { get; set; } = 7;

    [Range(1, 30)]
    public int RefreshTokenDays { get; set; } = 1;

    [Range(1, 1000)]
    public int LoginRequestsPerMinute { get; set; } = 20;

    [Range(1, 1000)]
    public int RefreshRequestsPerMinute { get; set; } = 60;

    /// <summary>
    /// Checks that the configured token signing key meets the required length.
    /// </summary>
    public bool HasValidSigningKey()
    {
        try
        {
            return Convert.FromBase64String(SigningKey).Length >= 32;
        }
        catch (FormatException)
        {
            return false;
        }
    }
}

```

StateReconciliationOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for state reconciliation.
/// </summary>
public sealed class StateReconciliationOptions
{
    public bool WorkerEnabled { get; set; } = true;
    [Range(1, 300)] public int PollSeconds { get; set; } = 5;
}

```

TenantAuthenticationOptions.cs

```

namespace BridgeControl.Options;

/// <summary>
/// Contains configuration settings for tenant authentication.
/// </summary>
public class TenantAuthenticationOptions : PlatformAuthenticationOptions
{
    public new const string SectionName = "Authentication:Tenant";
    /// <summary>
    /// Initializes a new <see cref="TenantAuthenticationOptions"/> instance and its
    dependencies.
    /// </summary>
    public TenantAuthenticationOptions() { Audience = "BridgeControl.Tenant"; }
}

```

VendorProvisioningOptions.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Options;

/// <summary>
/// Configures private local vendor storage and invitation delivery for the development host.
/// </summary>
public class VendorProvisioningOptions
{
    public const string SectionName = "VendorProvisioning";
    [Url]
    public string FrontendBaseUrl { get; set; } = "https://localhost:5173";
    public string StorageRoot { get; set; } =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"BridgeControl", "Storage");
    public string InvitationDirectory { get; set; } =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"BridgeControl", "Invitations");
    [Range(1, 168)]
    public int ActivationHours { get; set; } = 24;
    [Range(1, 1000)]
    public int ActivationRequestsPerMinute { get; set; } = 10;
}

```

Program.cs

```
using BridgeControl.Api;
using BridgeControl.Data;
using BridgeControl.Security;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Serilog;

bool setupAdministrator = args.Contains("--setup-platform-admin", StringComparison.Ordinal);
bool recoverAdministrator = args.Contains("--recover-platform-admin", StringComparison.Ordinal);
bool agentCredential = args.Contains("--agent-credential", StringComparison.Ordinal);
bool auditHistory = args.Contains("--audit-history", StringComparison.Ordinal);
bool vendorOperations = args.Contains("--vendor-operations", StringComparison.Ordinal);
bool backupDue = args.Contains("--backup-due", StringComparison.Ordinal);
if (new[] { setupAdministrator, recoverAdministrator, agentCredential, auditHistory,
vendorOperations, backupDue }.Count(value => value) > 1)
{
    Console.Error.WriteLine("Choose one maintenance command at a time.");
    Environment.ExitCode = 1;
    return;
}

WebApplicationBuilder builder = WebApplication.CreateBuilder(
    args.Where(argument => argument != "--setup-platform-admin" && argument !=
"--recover-platform-admin" && argument != "--agent-credential" && argument != "--audit-history"
&& argument != "--vendor-operations" && argument != "--backup-due").ToArray());

string? secretFile = Environment.GetEnvironmentVariable("BRIDGECONTROL_SECRETS_FILE");
if (!string.IsNullOrEmpty(secretFile))
{
    if (!Path.IsPathFullyQualified(secretFile)) { throw new InvalidOperationException("The
secret configuration file must use an absolute path."); }
    BridgeControl.Services.Shared.PrivateDirectory.Ensure(Path.GetDirectoryName(secretFile)!);
    VendorBackupStore.RejectLink(secretFile);
    builder.Configuration.AddJsonFile(secretFile, optional: false, reloadOnChange: false);
}

builder.Services.AddHostingProfile(builder.Configuration, builder.Environment);
builder.Services.AddWindowsService(options => options.ServiceName = "BridgeControl");

// Logging
builder.Services.AddSerilog((services, configuration) => configuration
    .MinimumLevel.Information()
    .MinimumLevel.Override("Microsoft.AspNetCore", Serilog.Events.LogEventLevel.Warning)
    .MinimumLevel.Override("Microsoft.EntityFrameworkCore.Database.Command",
Serilog.Events.LogEventLevel.Warning)
    .Enrich.FromLogContext()
    .WriteTo.Console()
    .WriteTo.File(
        Path.Combine(BridgeControl.Services.Shared.PrivateDirectory.Ensure(services.GetRequiredService<Mi
crosoft.Extensions.Options.IOptions<BridgeControl.Options.HostingProfileOptions>>().Value.LogDire
ctory), "bridgecontrol-.log"),
        rollingInterval: RollingInterval.Day,
        fileSizeLimitBytes: 10 * 1024 * 1024,
        rollOnFileSizeLimit: true,
        retainedFileCountLimit: 14));

// Database
string platformConnectionString =
    builder.Configuration.GetConnectionString("PlatformDatabase")
```



```

        ?? throw new InvalidOperationException(
            "The platform database connection string is missing.");

builder.Services.AddDbContext<PlatformDbContext>(options =>
    options.UseSqlServer(platformConnectionString));
builder.Services.AddHostedService<DatabaseHostLeaseService>();
builder.Services.AddOptions<BridgeControl.Options.DatabaseOperationsOptions>().Bind(builder.Configuration.GetSection("DatabaseOperations"));
builder.Services.AddScoped<IVendorDatabaseMaintenance, SqlVendorMaintenance>();
builder.Services.AddScoped<VendorBackupStore>();
builder.Services.AddScoped<VendorDatabaseOperations>();
builder.Services.AddScoped<DatabaseOperationsOperator>();

// API
builder.Services.AddApiConventions();
builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer<PlatformOpenApiTransformer>();
    options.AddDocumentTransformer<ApiAcceptanceDocumentation>();
});
builder.Services.AddPlatformAuthentication(builder.Configuration);
builder.Services.AddVendorProvisioning(builder.Configuration);
builder.Services.AddTenantAuthentication(builder.Configuration);
builder.Services.AddAgentAuthentication(builder.Configuration);

WebApplication app = builder.Build();

if (setupAdministrator || recoverAdministrator || agentCredential || auditHistory ||
vendorOperations || backupDue)
{
    await using AsyncServiceScope scope = app.Services.CreateAsyncScope();
    try
    {
        Environment.ExitCode = vendorOperations || backupDue
            ? await
scope.ServiceProvider.GetRequiredService<DatabaseOperationsOperator>().RunAsync(backupDue)
            : auditHistory
            ? await
scope.ServiceProvider.GetRequiredService<BridgeControl.Services.VendorAuditHistoryOperator>().RunInteractiveAsync()
            : agentCredential
            ? await
scope.ServiceProvider.GetRequiredService<AgentCredentialOperator>().RunInteractiveAsync()
            : setupAdministrator
            ? await
scope.ServiceProvider.GetRequiredService<PlatformAdminSetup>().RunInteractiveAsync()
            : await
scope.ServiceProvider.GetRequiredService<PlatformAdminRecovery>().RunInteractiveAsync();
    }
    catch (ArgumentException exception)
    {
        Console.Error.WriteLine(exception.Message);
        Environment.ExitCode = 1;
    }
    catch (Exception)
    {
        Console.Error.WriteLine("The maintenance command failed. Check the selected identity, database connection and applied migrations, then try again.");
        Environment.ExitCode = 1;
    }
    await app.DisposeAsync();
    return;
}

```

```

}

app.UseSerilogRequestLogging();

app.UseExceptionHandler();
app.UseStatusCodePages();
app.UseHostingProfile();

// API clients must use HTTPS before sending any credentials or authenticated requests.
app.Use(async (context, next) =>
{
    if (context.Request.Path.StartsWithSegments("/api") && !context.Request.IsHttps)
    {
        await Results.Problem(statusCode: StatusCodes.Status400BadRequest, title: "HTTPS
is required.").ExecuteAsync(context);
        return;
    }
    if (context.Request.Path.StartsWithSegments("/api/v1/platform") ||
context.Request.Path.StartsWithSegments("/api/v1/auth")
    || context.Request.Path.StartsWithSegments("/api/v1/agent") ||
context.Request.Path.StartsWithSegments("/api/v1/agent-enrollments"))
    {
        context.Response.Headers.CacheControl = "no-store";
        context.Response.Headers.Pragma = "no-cache";
    }
    await next(context);
});

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi().AllowAnonymous();
    app.UseSwaggerUI(options =>
    {
        options.RoutePrefix = "swagger";
        options.SwaggerEndpoint("../openapi/v1.json", "BridgeControl API v1");
        options.DocumentTitle = "BridgeControl API";
        options.ConfigObject.PersistAuthorization = false;
        options.ConfigObject.ValidatorUrl = null;
        // Swagger's parser requires a parenthesized parameter list when reconstructing
this function.
        options.UseRequestInterceptor("function (request) { request.credentials =
'same-origin'; return request; }");
    });
}
else
{
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRouting();
app.UseRateLimiter();
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();

app.Run();

/// <summary>
/// Exposes the application entry point to integration checks without changing normal startup.

```

```
/// </summary>
public partial class Program;
```

Repositories/AgentRepositories

AgentRepository.cs

```
using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.DTOS.Shared;
using BridgeControl.Models.AgentModels;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.AgentRepositories;
/// <summary>
/// Implements data access operations for agent.
/// </summary>
public sealed class AgentRepository(AgentIdentityService identity, TenantOperationFactory
operations,
    PlatformDbContext platform, ITenantResolver resolver, VendorSecurityAudit audit,
    TimeProvider clock) : IAgentRepository
{
    private static readonly Expression<Func<Agent, AgentResponse>> Response = x
        => new(x.Id, x.CustomerId, x.MachineName, x.AgentVersion, x.Status,
x.ReportedStatus, x.RegisteredAt, x.LastSeenAt);
    private static readonly Expression<Func<Installation, InstallationResponse>>
InstallationProjection = x
        => new(x.Id, x.CustomerId, x.ProductId, x.AgentId, x.Name, x.Status,
x.RegisteredAt, x.LastSeenAt);

    /// <summary>
    /// Creates an enrollment credential within the actor's permitted customer scope.
    /// </summary>
    public Task<AgentEnrollmentResponse> EnrollAsync(TenantActor actor,
CreateAgentEnrollmentRequest request, CancellationToken ct) => identity.EnrollAsync(actor,
request, ct);

    /// <summary>
    /// Registers an agent by consuming its enrollment credential.
    /// </summary>
    public Task<AgentRegistrationResponse> RegisterAsync(RegisterAgentRequest request,
CancellationToken ct) => identity.RegisterAsync(request, ct);

    /// <summary>
    /// Exchanges a machine credential for an agent access token.
    /// </summary>
```

```

        public Task<AgentTokenResponse> TokenAsync(AgentTokenRequest request, CancellationToken
ct) => identity.TokenAsync(request, ct);

        /// <summary>
        /// Builds the agent query within the caller's permitted scope.
        /// </summary>
        private static IQueryable<Agent> Scoped(TenantOperation operation)
        {
            VendorDbContext db = operation.Database;
            IQueryable<Customer> customers = operation.Access.Customers(db, "agents.read");
            IQueryable<Installation> installations = operation.Access.Installations(db,
"agents.read");
            bool installationScoped = operation.Access.User.CustomerId is not null &&
!operation.Access.Roles.Contains("CustomerAdministrator");
            return db.Agents.Where(x => customers.Any(customer => customer.Id == x.CustomerId)
&& (!installationScoped || installations.Any(installation =>
installation.AgentId == x.Id)));
        }

        /// <summary>
        /// Returns a filtered page of agent records.
        /// </summary>
        public async Task<PagedResponse<AgentResponse>> PageAsync(TenantActor actor,
PaginationQuery page, CancellationToken ct)
        {
            await using TenantOperation operation = await operations.OpenAsync(actor,
"agents.read", false, ct);
            return await new Repository<Agent, AgentResponse>(Scoped(operation),
Response).PageAsync(page, x => x.Id, ct);
        }

        /// <summary>
        /// Returns the requested agent record when available.
        /// </summary>
        public async Task<AgentResponse?> GetAsync(TenantActor actor, Guid id, CancellationToken
ct)
        {
            await using TenantOperation operation = await operations.OpenAsync(actor,
"agents.read", false, ct);
            return await new Repository<Agent, AgentResponse>(Scoped(operation).Where(x =>
x.Id == id), Response).SingleOrDefaultAsync(ct);
        }

        /// <summary>
        /// Returns a page of heartbeat history for an authorized agent.
        /// </summary>
        public async Task<PagedResponse<AgentHeartbeatResponse>> HeartbeatsAsync(TenantActor
actor, Guid id, PaginationQuery page, CancellationToken ct)
        {
            await using TenantOperation operation = await operations.OpenAsync(actor,
"agents.read", false, ct);
            if (!await Scoped(operation).AnyAsync(x => x.Id == id, ct)) { throw new
ApiAccessException(404); }
            return await new Repository<AgentHeartbeat,
AgentHeartbeatResponse>(operation.Database.AgentHeartbeats.Where(x => x.AgentId == id),
x => new(x.AgentVersion, x.Status, x.ReportedAt,
x.ReceivedAt)).PageAsync(page, x => x.Id, ct);
        }

        /// <summary>
        /// Records an ordered heartbeat and updates the agent's latest status.
        /// </summary>

```

```

        public async Task HeartbeatAsync(AgentActor actor, AgentHeartbeatRequest request,
CancellationToken ct)
        {
            if (request.ReportedAt == default || request.ReportedAt >
clock.UtcNow().AddMinutes(5))
            { throw new AgentValidationException("ReportedAt must be a real timestamp no more
than five minutes ahead of the server."); }
            await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
            await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
            Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);
            VendorDbContext db = tenant.Database;
            AgentHeartbeat? duplicate = await db.AgentHeartbeats.SingleOrDefaultAsync(x =>
x.AgentId == agent.Id && x.ReportedAt == request.ReportedAt, ct);
            if (duplicate is not null)
            {
                if (duplicate.AgentVersion != request.AgentVersion || duplicate.Status !=
request.Status)
                { throw new ApiConflictException("heartbeat-conflict", "A different
heartbeat already exists for this ReportedAt timestamp."); }
                return;
            }
            DateTimeOffset now = clock.UtcNow();
            db.AgentHeartbeats.Add(new() { AgentId = agent.Id, AgentVersion =
request.AgentVersion, Status = request.Status, ReportedAt = request.ReportedAt, ReceivedAt = now
});
            if (agent.LastReportedAt is null || request.ReportedAt > agent.LastReportedAt)
            {
                agent.LastReportedAt = request.ReportedAt; agent.LastSeenAt = now;
agent.AgentVersion = request.AgentVersion; agent.ReportedStatus = request.Status;
            }
            await db.SaveChangesAsync(ct);
        }

        /// <summary>
        /// Returns installation assignments available to the authenticated agent.
        /// </summary>
        public async Task<PagedResponse<InstallationResponse>> InstallationsAsync(AgentActor
actor, PaginationQuery page, Cancellation token ct)
        {
            await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
            Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);
            return await new Repository<Installation,
InstallationResponse>(tenant.Database.Installations.Where(x => x.AgentId == agent.Id
&& x.CustomerId == agent.CustomerId && x.Status == "Active"),
InstallationProjection).PageAsync(page, x => x.Id, ct);
        }

        /// <summary>
        /// Registers an installation using the agent's stable registration identity.
        /// </summary>
        public async Task<InstallationResponse> RegisterInstallationAsync(AgentActor actor,
RegisterInstallationRequest request, Cancellation token ct)
        {
            await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
            await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);

```

```

        Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        if (!await db.Products.AnyAsync(x => x.Id == request.ProductId && x.Status ==
"Active", ct)) { throw new ApiAccessException(404); }
        List<Installation> matches = await db.Installations.Where(x => x.CustomerId ==
agent.CustomerId && x.ProductId == request.ProductId
            && (x.RegistrationName == request.Name || (x.RegistrationName == null &&
x.Name == request.Name))).Take(2).ToListAsync(ct);
        if (matches.Count != 0)
        {
            if (matches.Count != 1 || matches[0].AgentId != agent.Id ||
matches[0].Status != "Active")
            { throw new ApiConflictException("installation-conflict", "This
installation cannot be registered by this Agent."); }
            return await db.Installations.Where(x => x.Id ==
matches[0].Id).Select(InstallationProjection).SingleAsync(ct);
        }
        Installation installation = new()
        {
            CustomerId = agent.CustomerId,
            AgentId = agent.Id,
            ProductId = request.ProductId,
            Name = request.Name,
            RegistrationName = request.Name,
            InstalledVersion = request.InstalledVersion,
            RegisteredAt = clock.GetUtcNow(),
            LastSeenAt = clock.GetUtcNow()
        };
        db.Installations.Add(installation);
        db.EffectiveConfigurationSignals.Add(new() { InstallationId = installation.Id,
CreatedAt = clock.GetUtcNow() });
        audit.Record(db, agent.Id, "AgentInstallationRegistered", installation.Id,
entityType: "Installation", actorType: "Agent");
        await db.SaveChangesAsync(ct);
        return await db.Installations.Where(x => x.Id ==
installation.Id).Select(InstallationProjection).SingleAsync(ct);
    }
}

```

IAgentRepository.cs

```

using BridgeControl.DTOs.AgentDTOs.AgentContracts;
using BridgeControl.DTOs.Shared;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.AgentRepositories;

/// <summary>
/// Defines data access operations for agent.
/// </summary>
public interface IAgentRepository
{
    /// <summary>
    /// Creates an enrollment credential within the actor's permitted customer scope.
    /// </summary>
    Task<AgentEnrollmentResponse> EnrollAsync(TenantActor actor, CreateAgentEnrollmentRequest
request, CancellationToken ct);
}

```

```

    /// <summary>
    /// Registers an agent by consuming its enrollment credential.
    /// </summary>
    Task<AgentRegistrationResponse> RegisterAsync(RegisterAgentRequest request,
CancellationTokens ct);

    /// <summary>
    /// Exchanges a machine credential for an agent access token.
    /// </summary>
    Task<AgentTokenResponse> TokenAsync(AgentTokenRequest request, CancellationTokens ct);

    /// <summary>
    /// Returns a filtered page of agent records.
    /// </summary>
    Task<PagedResponse<AgentResponse>> PageAsync(TenantActor actor, PaginationQuery page,
CancellationTokens ct);

    /// <summary>
    /// Returns the requested agent record when available.
    /// </summary>
    Task<AgentResponse?> GetAsync(TenantActor actor, Guid id, CancellationTokens ct);

    /// <summary>
    /// Returns a page of heartbeat history for an authorized agent.
    /// </summary>
    Task<PagedResponse<AgentHeartbeatResponse>> HeartbeatsAsync(TenantActor actor, Guid id,
PaginationQuery page, CancellationTokens ct);

    /// <summary>
    /// Records an ordered heartbeat and updates the agent's latest status.
    /// </summary>
    Task HeartbeatAsync(AgentActor actor, AgentHeartbeatRequest request, CancellationTokens
ct);

    /// <summary>
    /// Returns installation assignments available to the authenticated agent.
    /// </summary>
    Task<PagedResponse<InstallationResponse>> InstallationsAsync(AgentActor actor,
PaginationQuery page, CancellationTokens ct);

    /// <summary>
    /// Registers an installation using the agent's stable registration identity.
    /// </summary>
    Task<InstallationResponse> RegisterInstallationAsync(AgentActor actor,
RegisterInstallationRequest request, CancellationTokens ct);
}

```

Repositories/Platform

IPlatformAccountRepository.cs

```

using BridgeControl.DTOS.Platform;

namespace BridgeControl.Repositories.Platform;

/// <summary>

```

```

/// Defines the account lookup used after the platform controller has authenticated its caller.
/// </summary>
public interface IPlatformAccountRepository
{
    /// <summary>
    /// Returns the current administrator's public account data.
    /// </summary>
    Task<PlatformAdminDto> GetCurrentAsync(Guid authenticatedAccountId, CancellationToken
cancellationToken);
}

```

IVendorRepository.cs

```

using BridgeControl.DTOs.Platform;
using BridgeControl.DTOs.Shared;

namespace BridgeControl.Repositories.Platform;

/// <summary>
/// Provides platform-authorized vendor reads and audited name changes.
/// </summary>
public interface IVendorRepository
{
    /// <summary>
    /// Returns a filtered page of vendor records.
    /// </summary>
    Task<PagedResponse<VendorDto>> PageAsync(PaginationQuery pagination, CancellationToken
cancellationToken);

    /// <summary>
    /// Returns the requested vendor record when available.
    /// </summary>
    Task<VendorDto?> GetAsync(Guid vendorId, CancellationToken cancellationToken);

    /// <summary>
    /// Updates the supplied fields of the selected vendor record.
    /// </summary>
    Task<VendorDto?> UpdateAsync(Guid vendorId, UpdateVendorRequest request, Guid
administratorId, CancellationToken cancellationToken);
}

```

PlatformAccountRepository.cs

```

using BridgeControl.Data;
using BridgeControl.DTOs.Platform;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Platform;

/// <summary>
/// Reads only the platform account identified by the controller's validated identity.
/// </summary>
public sealed class PlatformAccountRepository(PlatformDbContext database) :
IPlatformAccountRepository
{
    /// <summary>
    /// Returns the current administrator's public account data.

```



```

    /// </summary>
    public Task<PlatformAdminDto> GetCurrentAsync(Guid authenticatedAccountId,
CancellationToken cancellationToken)
    {
        return database.PlatformAdmins.AsNoTracking()
            .Where(account => account.Id == authenticatedAccountId)
            .Select(account => new PlatformAdminDto
            {
                Id = account.Id,
                Email = account.Email,
                Status = account.Status,
                CreatedAt = account.CreatedAt,
                LastLoginAt = account.LastLoginAt
            })
            .SingleOrDefaultAsync(cancellationToken);
    }
}

```

VendorRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Platform;
using BridgeControl.DTOs.Shared;
using BridgeControl.Models.Platform;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Platform;

/// <summary>
/// Projects safe vendor responses and changes only supplied, editable fields.
/// </summary>
public class VendorRepository(PlatformDbContext database, TimeProvider clock) : IVendorRepository
{
    private static readonly Expression<Func<SoftwareVendor, VendorDto>> Response = vendor =>
new VendorDto
    {
        Id = vendor.Id,
        Name = vendor.Name,
        Status = vendor.Status,
        CreatedAt = vendor.CreatedAt,
        DeactivatedAt = vendor.DeactivatedAt
    };

    /// <summary>
    /// Returns a filtered page of vendor records.
    /// </summary>
    public Task<PagedResponse<VendorDto>> PageAsync(PaginationQuery pagination,
CancellationToken cancellationToken)
    {
        => new Repository<SoftwareVendor, VendorDto>(database.SoftwareVendors,
Response).PageAsync(pagination, vendor => vendor.Id, cancellationToken);
    }

    /// <summary>
    /// Returns the requested vendor record when available.
    /// </summary>
    public Task<VendorDto?> GetAsync(Guid vendorId, CancellationToken cancellationToken)

```

```

=> new Repository<SoftwareVendor, VendorDto>(database.SoftwareVendors.Where(vendor
=> vendor.Id == vendorId), Response).SingleOrDefaultAsync(cancellationToken);

    /// <summary>
    /// Updates the supplied fields of the selected vendor record.
    /// </summary>
    public async Task<VendorDto?> UpdateAsync(Guid vendorId, UpdateVendorRequest request, Guid
administratorId, CancellationToken cancellationToken)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(database, cancellationToken);
        SoftwareVendor? vendor = await
database.SoftwareVendors.SingleOrDefaultAsync(vendor => vendor.Id == vendorId,
cancellationToken);
        if (vendor is null) { return null; }
        if (request.NameProvided && request.Name != vendor.Name)
        {
            if (await database.SoftwareVendors.AnyAsync(other => other.Id != vendorId
&& other.Name == request.Name, cancellationToken))
            {
                throw new ApiConflictException("vendor-name-in-use", "A vendor with
that name already exists.");
            }
            // PATCH only changes fields that were supplied; MapDto also copies omitted
null/default values.
            vendor.Name = request.Name!;
            database.PlatformAuditLogs.Add(new PlatformAuditLog
            {
                PlatformAdminId = administratorId,
                VendorId = vendor.Id,
                Action = "VendorRenamed",
                EntityType = nameof(SoftwareVendor),
                EntityId = vendor.Id.ToString(),
                Details = "{}",
                CreatedAt = clock.GetUtcNow()
            });
            await database.SaveChangesAsync(cancellationToken);
        }
        return await GetAsync(vendorId, cancellationToken);
    }
}

```

Repositories/Shared

IRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.DTOS.Shared;

namespace BridgeControl.Repositories.Shared;

    /// <summary>
    /// Defines the shared asynchronous reads available from an already authorized query.
    /// </summary>
    public interface IRepository<TEntity, TResponse>
        where TEntity : class

```

```

        where TResponse : class
    {
        /// <summary>
        /// Returns exactly one response, failing if the query matches zero or multiple records.
        /// </summary>
        Task<TResponse> SingleAsync(CancellationTokentoken cancellationToken = default);

        /// <summary>
        /// Returns one response or null, failing if the query matches multiple records.
        /// </summary>
        Task<TResponse?> SingleOrDefaultAsync(CancellationTokentoken cancellationToken = default);

        /// <summary>
        /// Returns a bounded page using a unique ordering key and the same access filters for its
total count.
        /// </summary>
        Task<PagedResponse<TResponse>> PageAsync<TKey>(PaginationQuery pagination,
            Expression<Func<TEntity, TKey>> uniqueOrderKey, CancellationTokentoken
cancellationToken = default);
    }
}

```

Repository.cs

```

using System.Linq.Expressions;
using BridgeControl.DTOs.Shared;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Shared;

/// <summary>
/// Reads DTOs from a caller-authorized query without choosing a database, broadening access or
exposing tracked entities.
/// </summary>
public sealed class Repository<TEntity, TResponse> : IRepository<TEntity, TResponse>
    where TEntity : class
    where TResponse : class
{
    private readonly IQueryable<TEntity> scopedQuery;
    private readonly Expression<Func<TEntity, TResponse>> projection;

    /// <summary>
    /// Initializes the repository with its source query and response projection.
    /// </summary>
    public Repository(IQueryable<TEntity> scopedQuery, Expression<Func<TEntity, TResponse>>
projection)
    {
        ArgumentNullException.ThrowIfNull(scopedQuery);
        ArgumentNullException.ThrowIfNull(projection);
        this.scopedQuery = scopedQuery.AsNoTracking();
        this.projection = projection;
    }

    /// <summary>
    /// Executes the mapped query and requires exactly one result.
    /// </summary>
    public Task<TResponse> SingleAsync(CancellationTokentoken cancellationToken = default)
        => scopedQuery.Select(projection).SingleAsync(cancellationToken);

    /// <summary>

```

```

        /// Executes the mapped query and returns its single result or the default value.
        /// </summary>
        public Task<TResponse?> SingleOrDefaultAsync(CancellationTokentoken cancellationToken =
default)
            => scopedQuery.Select(projection).SingleOrDefaultAsync(cancellationToken);

        /// <summary>
        /// Pages and counts the same authorized rows; the ordering key must uniquely identify
each row.
        /// </summary>
        public Task<PagedResponse<TResponse>> PageAsync<TKey>(PaginationQuery pagination,
            Expression<Func<TEntity, TKey>> uniqueOrderKey, CancellationTokentoken
cancellationToken = default)
            => scopedQuery.ToPageAsync(pagination, uniqueOrderKey, projection,
cancellationToken);
    }

```

Repositories/Vendor

AccessAssignmentRepository.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.UserAdministrationRequests;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Validates assignment authority and saves role or resource grants with their audit events.
/// </summary>
public class AccessAssignmentRepository(TenantOperationFactory operations, RoleGrantPolicy
policy, VendorSecurityAudit audit, TimeProvider clock) : IAccessAssignmentRepository
{
    /// <summary>
    /// Lists the roles the current actor is allowed to assign.
    /// </summary>
    public async Task<PagedResponse<RoleDto>> RolesAsync(TenantActor actor, PaginationQuery
page, CancellationTokentoken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.read", false, ct);
        return await new Repository<Role, RoleDto>(policy.Assignable(operation.Database,
operation.Access),
            role => new RoleDto { Id = role.Id, Name = role.Name, Scope = role.Scope })
            .PageAsync(page, role => role.Id, ct);
    }

    /// <summary>
    /// Lists permissions belonging to roles the current actor is allowed to assign.

```

```

    /// </summary>
    public async Task<PagedResponse<PermissionDto>> PermissionsAsync(TenantActor actor,
        PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.read", false, ct);
        VendorDbContext database = operation.Database;
        IQueryable<Guid> assignableRoleIds = policy.Assignable(database,
operation.Access).Select(role => role.Id);
        IQueryable<Permission> query = database.Permissions.Where(permission =>
database.RolePermissions
            .Any(grant => grant.PermissionId == permission.Id &&
assignableRoleIds.Contains(grant.RoleId)));
        return await new Repository<Permission, PermissionDto>(query,
            permission => new PermissionDto { Id = permission.Id, Code =
permission.Code, Description = permission.Description })
            .PageAsync(page, permission => permission.Id, ct);
    }

    /// <summary>
    /// Replaces the user's role assignments after checking the actor's grant authority.
    /// </summary>
    public async Task ReplaceRolesAsync(TenantActor actor, Guid userId, SetUserRolesRequest
request, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.manage", true, ct);
        VendorDbContext database = operation.Database;
        UserAccount user = await SubjectAsync(operation, userId, "users.manage", ct);
        List<Role> desiredRoles = await policy.ValidateAsync(database, operation.Access,
user.CustomerId, request.RoleIds, ct);
        if (user.UserRoles.Select(grant =>
grant.RoleId).ToHashSet().SetEquals(request.RoleIds))
        {
            return;
        }

        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(ct);
        UserRole[] removedRoles = user.UserRoles.Where(grant =>
!request.RoleIds.Contains(grant.RoleId)).ToArray();
        foreach (UserRole grant in removedRoles)
        {
            audit.Record(database, actor.UserId, "UserRoleRemoved", userId, entityType:
"UserRole", resourceId: grant.RoleId);
        }
        database.UserRoles.RemoveRange(removedRoles);
        foreach (Role role in desiredRoles.Where(role => !user.UserRoles.Any(grant =>
grant.RoleId == role.Id)))
        {
            database.UserRoles.Add(new()
            {
                Id = Guid.NewGuid(),
                UserId = userId,
                RoleId = role.Id
            });
            audit.Record(database, actor.UserId, "UserRoleGranted", userId, entityType:
"UserRole", resourceId: role.Id);
        }

        // Role changes remove grants that the new roles can no longer use.
        bool canKeepCustomerGrants = IsTechnician(desiredRoles);

```

```

        bool canKeepInstallationGrants = IsOrdinaryCustomerUser(desiredRoles);
        if (!canKeepCustomerGrants)
        {
            List<TechnicianCustomerAccess> customerGrants = await
database.TechnicianCustomerAccessGrants
                .Where(grant => grant.UserId == userId)
                .ToListAsync(ct);
            foreach (TechnicianCustomerAccess grant in customerGrants)
            {
                database.Remove(grant);
                audit.Record(database, actor.UserId, "CustomerAccessRemoved",
userId, resourceId: grant.CustomerId);
            }
        }
        if (!canKeepInstallationGrants)
        {
            List<CustomerUserInstallationAccess> installationGrants = await
database.CustomerUserInstallationAccessGrants
                .Where(grant => grant.UserId == userId)
                .ToListAsync(ct);
            foreach (CustomerUserInstallationAccess grant in installationGrants)
            {
                database.Remove(grant);
                audit.Record(database, actor.UserId, "InstallationAccessRemoved",
userId, resourceId: grant.InstallationId);
            }
        }

        // Updating the user also checks its row version for concurrent assignment
changes.
        database.Entry(user).Property(account => account.Status).IsModified = true;
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
    }

    /// <summary>
    /// Lists the customer's access assignments for the selected technician.
    /// </summary>
    public async Task<SetCustomerAccessRequest> CustomersAsync(TenantActor actor, Guid userId,
CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.read", false, ct);
        await TechnicianAsync(operation, userId, "users.read", ct);
        Guid[] customerIds = await operation.Database.TechnicianCustomerAccessGrants
            .Where(grant => grant.UserId == userId)
            .OrderBy(grant => grant.CustomerId)
            .Select(grant => grant.CustomerId)
            .ToArrayAsync(ct);
        return new() { CustomerIds = customerIds };
    }

    /// <summary>
    /// Replaces the customer's access assignments for the selected technician.
    /// </summary>
    public async Task ReplaceCustomersAsync(TenantActor actor, Guid userId,
SetCustomerAccessRequest request, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.manage", true, ct);
        UserAccount user = await TechnicianAsync(operation, userId, "users.manage", ct);
        VendorDbContext database = operation.Database;

```

```

        int accessibleCustomerCount = await operation.Access.Customers(database)
            .CountAsync(customer => request.CustomerIds.Contains(customer.Id), ct);
        if (accessibleCustomerCount != request.CustomerIds.Length)
        {
            throw new ApiAccessException(404);
        }
        List<TechnicianCustomerAccess> existingGrants = await
            database.TechnicianCustomerAccessGrants
                .Where(grant => grant.UserId == userId)
                .ToListAsync(ct);
        if (existingGrants.Select(grant =>
            grant.CustomerId).ToHashSet().SetEquals(request.CustomerIds))
        {
            return;
        }

        await using IDbContextTransaction transaction = await
            database.Database.BeginTransactionAsync(ct);
        foreach (TechnicianCustomerAccess grant in existingGrants.Where(grant =>
            !request.CustomerIds.Contains(grant.CustomerId)))
        {
            database.Remove(grant);
            audit.Record(database, actor.UserId, "CustomerAccessRemoved", userId,
                resourceId: grant.CustomerId);
        }
        foreach (Guid customerId in request.CustomerIds.Where(customerId =>
            !existingGrants.Any(grant => grant.CustomerId == customerId)))
        {
            database.TechnicianCustomerAccessGrants.Add(new()
            {
                Id = Guid.NewGuid(),
                UserId = userId,
                CustomerId = customerId,
                GrantedByUserId = actor.UserId,
                GrantedAt = clock.GetUtcNow()
            });
            audit.Record(database, actor.UserId, "CustomerAccessGranted", userId,
                resourceId: customerId);
        }

        // Updating the user also checks its row version for concurrent assignment
        changes.
        database.Entry(user).Property(account => account.Status).IsModified = true;
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
    }

    /// <summary>
    /// Lists installation access assignments for the selected user.
    /// </summary>
    public async Task<SetInstallationAccessRequest> InstallationsAsync(TenantActor actor, Guid
        userId, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
            "users.read", false, ct);
        await OrdinaryUserAsync(operation, userId, "users.read", ct);
        Guid[] installationIds = await
            operation.Database.CustomerUserInstallationAccessGrants
                .Where(grant => grant.UserId == userId)
                .OrderBy(grant => grant.InstallationId)
                .Select(grant => grant.InstallationId)
                .ToArrayAsync(ct);
    }

```

```

        return new() { InstallationIds = installationIds };
    }

    /// <summary>
    /// Replaces installation access assignments for the selected user.
    /// </summary>
    public async Task ReplaceInstallationsAsync(TenantActor actor, Guid userId,
        SetInstallationAccessRequest request, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.manage", true, ct);
        UserAccount user = await OrdinaryUserAsync(operation, userId, "users.manage", ct);
        VendorDbContext database = operation.Database;
        int accessibleInstallationCount = await operation.Access.Installations(database)
            .CountAsync(installation => installation.CustomerId == user.CustomerId
                && request.InstallationIds.Contains(installation.Id), ct);
        if (accessibleInstallationCount != request.InstallationIds.Length)
        {
            throw new ApiAccessException(404);
        }
        List<CustomerUserInstallationAccess> existingGrants = await
database.CustomerUserInstallationAccessGrants
            .Where(grant => grant.UserId == userId)
            .ToListAsync(ct);
        if (existingGrants.Select(grant =>
grant.InstallationId).ToHashSet().SetEquals(request.InstallationIds))
        {
            return;
        }

        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(ct);
        foreach (CustomerUserInstallationAccess grant in existingGrants.Where(grant =>
!request.InstallationIds.Contains(grant.InstallationId)))
        {
            database.Remove(grant);
            audit.Record(database, actor.UserId, "InstallationAccessRemoved", userId,
resourceId: grant.InstallationId);
        }
        foreach (Guid installationId in request.InstallationIds.Where(installationId =>
!existingGrants.Any(grant => grant.InstallationId == installationId)))
        {
            database.CustomerUserInstallationAccessGrants.Add(new()
            {
                Id = Guid.NewGuid(),
                UserId = userId,
                CustomerId = user.CustomerId!.Value,
                InstallationId = installationId,
                GrantedByUserId = actor.UserId,
                GrantedAt = clock.GetUtcNow()
            });
            audit.Record(database, actor.UserId, "InstallationAccessGranted", userId,
resourceId: installationId);
        }

        // Updating the user also checks its row version for concurrent assignment
        changes.
        database.Entry(user).Property(account => account.Status).IsModified = true;
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
    }

```



```

    /// <summary>
    /// Loads the assignment subject within the actor's permitted scope.
    /// </summary>
    private static async Task<UserAccount> SubjectAsync(TenantOperation operation, Guid
userId, string permission, CancellationToken ct)
        => await operation.Access.Users(operation.Database, permission)
            .Include(user => user.UserRoles)
            .ThenInclude(grant => grant.Role)
            .SingleOrDefaultAsync(user => user.Id == userId, ct)
            ?? throw new ApiAccessException(404);

    /// <summary>
    /// Requires the assignment subject to be a technician.
    /// </summary>
    private static async Task<UserAccount> TechnicianAsync(TenantOperation operation, Guid
userId, string permission, CancellationToken ct)
    {
        if (!RoleGrantPolicy.VendorAdministrator(operation.Access))
        {
            throw new ApiAccessException(403);
        }
        UserAccount user = await SubjectAsync(operation, userId, permission, ct);
        if (user.CustomerId is not null || !IsTechnician(user.UserRoles.Select(grant =>
grant.Role)))
        {
            throw new ApiConflictException("technician-required", "Customer assignments
require a vendor technician.");
        }
        return user;
    }

    /// <summary>
    /// Requires the assignment subject to be an ordinary customer user.
    /// </summary>
    private static async Task<UserAccount> OrdinaryUserAsync(TenantOperation operation, Guid
userId, string permission, CancellationToken ct)
    {
        if (!RoleGrantPolicy.VendorAdministrator(operation.Access) &&
!RoleGrantPolicy.CustomerAdministrator(operation.Access))
        {
            throw new ApiAccessException(403);
        }
        UserAccount user = await SubjectAsync(operation, userId, permission, ct);
        if (user.CustomerId is null || !IsOrdinaryCustomerUser(user.UserRoles.Select(grant
=> grant.Role)))
        {
            throw new ApiConflictException("customer-user-required", "Installation
assignments require an ordinary customer user.");
        }
        return user;
    }

    private static bool IsTechnician(IEnumerable<Role> roles)
        => roles.Any(role => role.Name == "VendorTechnician")
            && !roles.Any(role => role.Name == "VendorAdministrator");

    private static bool IsOrdinaryCustomerUser(IEnumerable<Role> roles)
        => roles.Any(role => role.Name == "CustomerUser")
            && !roles.Any(role => role.Name == "CustomerAdministrator");
}

```

AgentArtifactContent.cs

```
using BridgeControl.Models.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Represents agent artifact content data.
/// </summary>
public sealed record AgentArtifactContent(Stream Stream, FileArtifact Artifact, Guid
ProductVersionId, Guid ConfigurationRevisionId);
```

AgentDesiredState.cs

```
using BridgeControl.DTOs.Vendor.InstallationStateContracts;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Represents agent desired state data.
/// </summary>
public sealed record AgentDesiredState(DesiredStateResponse State, Guid InstallerArtifactId, Guid
ConfigurationArtifactId);
```

ConfigurationMigrationExecutionRepository.cs

```
using System.Text.Json;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.ConfigurationMigrationExecution;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for configuration migration execution.
/// </summary>
public sealed class ConfigurationMigrationExecutionRepository(ConfigurationMigrationService
service) : IConfigurationMigrationExecutionRepository
{
    /// <summary>
    /// Creates a configuration migration preview for the authorized installation.
    /// </summary>
    public async Task<ConfigurationMigrationPreviewResponse> PreviewAsync(TenantActor actor,
Guid installationId, ConfigurationMigrationPreviewRequest request, CancellationToken ct)
    {
        ConfigurationMigrationReview review = await service.ReviewAsync(actor,
installationId, request.TargetProductVersionId, ct);
        ConfigurationMigrationPlan plan =
JsonSerializer.Deserialize<ConfigurationMigrationPlan>(review.PlanJson)!;
        return new(plan.TargetProductVersionId, plan.Values, plan.Errors.Count == 0,
plan.Errors,
[.. plan.Changes.Select(x => new
ConfigurationMigrationChangeResponse(x.RuleId, x.Layer, x.RuleType, x.SourcePath, x.TargetPath,
x.IsDestructive, x.PreviousValueHash))]);
    }
}
```

```

    /// <summary>
    /// Applies an approved configuration migration for the authorized installation.
    /// </summary>
    public Task<EffectiveConfigurationResponse> ApplyAsync(TenantActor actor, Guid
installationId, ApplyConfigurationMigrationRequest request, CancellationToken ct)
        => service.ApplyLatestAsync(actor, installationId, request.TargetProductVersionId,
request.ApprovedDestructiveRuleIds, ct);
}

```

ConfigurationMigrationRepository.cs

```

using System.Linq.Expressions;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for configuration migration.
/// </summary>
public sealed class ConfigurationMigrationRepository(TenantOperationFactory operations,
ConfigurationMigrationRules rules, VendorSecurityAudit audit, TimeProvider clock) :
IConfigurationMigrationRepository
{
    private static readonly Expression<Func<ConfigurationMigration,
ConfigurationMigrationDto>> Projection = x => new()
    {
        Id = x.Id,
        FromProductVersionId = x.FromProductVersionId,
        ToProductVersionId = x.ToProductVersionId,
        CreatedAt = x.CreatedAt,
        CreatedByUserId = x.CreatedByUserId,
        Rules = x.Rules.OrderBy(r => r.Sequence).Select(r => new MigrationRuleDto
        {
            Id = r.Id,
            MigrationId = r.MigrationId,
            Sequence = r.Sequence,
            RuleType = r.RuleType,
            SourcePath = r.SourcePath,
            TargetPath = r.TargetPath,
            Transform = r.Transform,
            IsDestructive = r.IsDestructive
        }).ToList();
    };

    /// <summary>
    /// Builds the query for configuration migration records.
    /// </summary>
    private static IQueryable<ConfigurationMigration> Query(TenantOperation op, Guid product)
=> op.Database.ConfigurationMigrations
        .Where(x => x.FromProductVersion.ProductId == product &&
x.ToProductVersion.ProductId == product);

```

```

    /// <summary>
    /// Returns the matching configuration migration records.
    /// </summary>
    public async Task<PagedResponse<ConfigurationMigrationDto>> ListAsync(TenantActor actor,
Guid product, PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        if (!await op.Database.Products.AnyAsync(x => x.Id == product, ct)) { throw new
ApiAccessException(404); }
        return await new Repository<ConfigurationMigration,
ConfigurationMigrationDto>(Query(op, product), Projection).PageAsync(page, x => x.Id, ct);
    }

    /// <summary>
    /// Returns the requested configuration migration record when available.
    /// </summary>
    public async Task<ConfigurationMigrationDto?> GetAsync(TenantActor actor, Guid product,
Guid id, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        return await new Repository<ConfigurationMigration,
ConfigurationMigrationDto>(Query(op, product).Where(x => x.Id == id),
Projection).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Validates and persists the configuration migration definition.
    /// </summary>
    public async Task<ConfigurationMigrationDto> SaveAsync(TenantActor actor, Guid product,
Guid? id, CreateConfigurationMigrationRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.manage", true, ct);
        if (!RoleGrantPolicy.VendorAdministrator(op.Access)) { throw new
ApiAccessException(403); }
        rules.Validate(request.Rules);
        VendorDbContext db = op.Database;
        if (request.FromProductVersionId == request.ToProductVersionId) { throw new
ConfigurationValidationException("Source and target versions must differ."); }
        if (await db.ProductVersions.CountAsync(x => x.ProductId == product && (x.Id ==
request.FromProductVersionId || x.Id == request.ToProductVersionId), ct) != 2)
        { throw new ApiAccessException(404); }
        ConfigurationMigration migration = id is null ? new ConfigurationMigration
        {
            Id = Guid.NewGuid(),
            FromProductVersionId = request.FromProductVersionId,
            ToProductVersionId = request.ToProductVersionId,
            CreatedByUserId = actor.UserId,
            CreatedAt = clock.GetUtcNow()
        }
        : await Query(op, product).Include(x => x.Rules).SingleOrDefaultAsync(x =>
x.Id == id, ct) ?? throw new ApiAccessException(404);
        if (migration.FromProductVersionId != request.FromProductVersionId ||
migration.ToProductVersionId != request.ToProductVersionId)
        { throw new ApiConflictException("migration-ownership", "Replace rules without
changing the migration's version pair."); }
        var graph = await Query(op, product).Where(x => x.Id != migration.Id).Select(x =>
new { x.Id, From = x.FromProductVersionId, To = x.ToProductVersionId
}).Take(257).ToListAsync(ct);
    }

```

```

        if (graph.Count >= 256) { throw new ConfigurationValidationException("A product
supports at most 256 migration definitions."); }
        if (graph.Any(x => x.From == request.FromProductId && x.To ==
request.ToProductId))
        { throw new ApiConflictException("migration-exists", "A migration already exists
for this version pair."); }
        HashSet<Guid> visited = []; Queue<Guid> remaining = new();
remaining.Enqueue(request.ToProductId);
        while (remaining.TryDequeue(out Guid next))
        {
            if (next == request.FromProductId) { throw new
ApiConflictException("migration-cycle", "Migration definitions must not form cycles."); }
            if (visited.Add(next)) { foreach (var edge in graph.Where(x => x.From ==
next)) { remaining.Enqueue(edge.To); } }
        }
        MigrationRuleRequest[] normalized = request.Rules.OrderBy(x =>
x.Sequence).Select(x => x with { SourcePath = x.SourcePath ?? "", TargetPath = x.TargetPath ??
"", Transform = x.Transform ?? "" }).ToArray();
        MigrationRuleRequest[] previous = migration.Rules.OrderBy(x =>
x.Sequence).Select(x => new MigrationRuleRequest(x.Sequence, x.RuleType, x.SourcePath,
x.TargetPath, x.Transform, x.IsDestructive)).ToArray();
        if (id is not null && migration.RevisionNumber > 0 &&
previous.SequenceEqual(normalized))
        { return await Query(op, product).Where(x => x.Id ==
migration.Id).Select(Projection).SingleOrDefaultAsync(ct); }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        if (id is null) { db.ConfigurationMigrations.Add(migration); }
        else { db.MigrationRules.RemoveRange(migration.Rules); migration.Rules.Clear();
await db.SaveChangesAsync(ct); }
        migration.RevisionNumber = checked(migration.RevisionNumber + 1);
        foreach (MigrationRuleRequest? rule in normalized)
        {
            MigrationRule entity = new MigrationRule
            {
                Id = Guid.NewGuid(),
                MigrationId = migration.Id,
                Sequence = rule.Sequence,
                RuleType = rule.RuleType,
                SourcePath = rule.SourcePath!,
                TargetPath = rule.TargetPath!,
                Transform = rule.Transform!,
                IsDestructive = rule.IsDestructive
            };
            migration.Rules.Add(entity); db.MigrationRules.Add(entity);
        }
        MigrationRuleDto[] snapshot = migration.Rules.OrderBy(x => x.Sequence).Select(x =>
new MigrationRuleDto
        {
            Id = x.Id,
            MigrationId = migration.Id,
            Sequence = x.Sequence,
            RuleType = x.RuleType,
            SourcePath = x.SourcePath,
            TargetPath = x.TargetPath,
            Transform = x.Transform,
            IsDestructive = x.IsDestructive
        }).ToArray();
        db.MigrationDefinitionRevisions.Add(new() { MigrationId = migration.Id,
RevisionNumber = migration.RevisionNumber, RulesJson = JsonSerializer.Serialize(snapshot),
CreatedByUserId = actor.UserId, CreatedAt = clock.UtcNow() });

```

```

        audit.Record(db, actor.UserId, id is null ? "ConfigurationMigrationCreated" :
"ConfigurationMigrationReplaced", migration.Id, entityType: "ConfigurationMigration");
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
        return await Query(op, product).Where(x => x.Id ==
migration.Id).Select(Projection).SingleAsync(ct);
    }
}

```

ConfigurationRepository.cs

```

using System.Security.Cryptography;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for configuration.
/// </summary>
public sealed class ConfigurationRepository(TenantOperationFactory operations,
ConfigurationValues values,
ConfigurationArtifactReader reader, VendorArtifactArchive archive, VendorSecurityAudit
audit, TimeProvider clock) : IConfigurationRepository
{
    /// <summary>
    /// Returns the current configuration revision.
    /// </summary>
    public async Task<ConfigurationRevisionResponse?> CurrentAsync(TenantActor actor,
ConfigurationScope scope, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        await AuthorizeAsync(op, scope, false, ct);
        ConfigurationRevision? revision = await Revisions(op, scope).Include(x =>
x.FileArtifact).OrderByDescending(x => x.RevisionNumber).FirstOrDefaultAsync(ct);
        return revision is null ? null : await ResponseAsync(op, revision, ct);
    }

    /// <summary>
    /// Returns a page of historical configuration revisions.
    /// </summary>
    public async Task<PagedResponse<ConfigurationRevisionResponse>> HistoryAsync(TenantActor
actor, ConfigurationScope scope, PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        await AuthorizeAsync(op, scope, false, ct);
        IQueryable<ConfigurationRevision> query = Revisions(op, scope).AsNoTracking();
        PagedResponse<ConfigurationRevision> revisions = await query.Include(x =>
x.FileArtifact)
            .ToPageAsync(page, x => x.RevisionNumber, ct, descending: true);
    }
}

```

```

        return await revisions.MapAsync((revision, token) => ResponseAsync(op, revision,
token), ct);
    }

    /// <summary>
    /// Writes a validated configuration revision at the requested scope.
    /// </summary>
    public async Task<ConfigurationRevisionResponse> SetAsync(TenantActor actor,
ConfigurationScope scope, UpdateConfigurationRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.manage", true, ct);
        Guid productId = await AuthorizeAsync(op, scope, true, ct);
        VendorDbContext db = op.Database;
        ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x => x.Id
== request.ProductVersionId && x.ProductId == productId, ct) ?? throw new
ApiAccessException(404);
        FileArtifact? schemaArtifact = await db.ProductVersionArtifacts.Where(x =>
x.ProductVersionId == version.Id && x.ArtifactType == "Schema").Select(x =>
x.FileArtifact).SingleOrDefaultAsync(ct);
        if (schemaArtifact is null) { throw new ApiConflictException("schema-required",
"Attach a valid schema before editing configuration."); }
        if (version.Status != "Draft" && (version.Status != "Published" ||
version.PublishedSchemaArtifactId != schemaArtifact.Id))
        { throw new ApiConflictException("version-unavailable", "The version does not have
a verified published schema."); }
        JsonElement schema = await reader.ReadAsync(op.Tenant.ArchivePath, schemaArtifact,
true, ct);

        byte[] canonical = values.Normalize(request.Values);
        using JsonDocument document = JsonDocument.Parse(canonical);
        values.Validate(schema, document.RootElement, ct);
        string hash = Convert.ToHexStringLower(SHA256.HashData(canonical));
        ConfigurationProfile? profile = await Profiles(op,
scope).SingleOrDefaultAsync(ct);
        ConfigurationRevision? currentForVersion = profile is null ? null : await
db.ConfigurationRevisions.Include(x => x.FileArtifact)
            .Where(x => x.ConfigurationProfileId == profile.Id && x.ProductVersionId ==
version.Id).OrderByDescending(x => x.RevisionNumber).FirstOrDefaultAsync(ct);
        if (currentForVersion is not null && currentForVersion.ValidationStatus == "Valid"
&& currentForVersion.SchemaArtifactId == schemaArtifact.Id
            && currentForVersion.FileArtifact.Sha256 == hash &&
currentForVersion.FileArtifact.SizeBytes == canonical.Length)
        { return await ResponseAsync(op, currentForVersion, ct); }
        bool productDefault = scope.CustomerId is null && scope.InstallationId is null;
        if (productDefault) { ProductRepository.RequireDraft(version); }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        if (profile is null)
        {
            profile = new() { Id = Guid.NewGuid(), ProductId = scope.ProductId,
CustomerId = scope.CustomerId, InstallationId = scope.InstallationId, CreatedAt =
clock.GetUtcNow() };
            db.ConfigurationProfiles.Add(profile);
        }
        int revisionNumber = checked(profile.LastRevisionNumber + 1);
        profile.LastRevisionNumber = revisionNumber;
        // Touch the version to detect a concurrent schema/release write from a stale
external server snapshot.
        db.Entry(version).Property(x => x.Version).IsModified = true;
        await using StagedArtifact staged = await
archive.StageConfigurationAsync(op.Tenant.ArchivePath, canonical, ct);
        FileArtifact artifact = new()

```

```

        {
            Id = Guid.NewGuid(),
            ArtifactType = "Configuration",
            FileName = staged.FileName,
            ContentType = staged.ContentType,
            StorageIdentifier = staged.StorageIdentifier,
            Sha256 = staged.Sha256,
            SizeBytes = staged.SizeBytes,
            CreatedAt = clock.GetUtcNow()
        };
        ConfigurationRevision revision = new()
        {
            Id = Guid.NewGuid(),
            ConfigurationProfileId = profile.Id,
            ProductVersionId = version.Id,
            RevisionNumber = revisionNumber,
            FileArtifact = artifact,
            SchemaArtifactId = schemaArtifact.Id,
            ValidationStatus = "Valid",
            CreatedByUserId = actor.UserId,
            CreatedAt = clock.GetUtcNow()
        };
        db.ConfigurationRevisions.Add(revision);
        db.ConfigurationSourceChanges.Add(new() { Id = Guid.NewGuid(),
ConfigurationRevisionId = revision.Id, CreatedAt = clock.GetUtcNow() });
        audit.Record(db, actor.UserId, "ConfigurationRevised", profile.Id, entityType:
"ConfigurationProfile", resourceId: revision.Id);
        staged.Promote();
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
        return new(revision.Id, version.Id, revisionNumber, document.RootElement.Clone(),
actor.UserId, revision.CreatedAt);
    }

    /// <summary>
    /// Selects configuration profiles at the requested product, customer, or installation
    scope.
    /// </summary>
    private static IQueryable<ConfigurationProfile> Profiles(TenantOperation op,
ConfigurationScope scope)
        => op.Database.ConfigurationProfiles.Where(x => x.ProductId == scope.ProductId &&
x.CustomerId == scope.CustomerId && x.InstallationId == scope.InstallationId);

    /// <summary>
    /// Selects immutable revisions belonging to the requested configuration scope.
    /// </summary>
    private static IQueryable<ConfigurationRevision> Revisions(TenantOperation op,
ConfigurationScope scope)
        => Profiles(op, scope).SelectMany(x => x.Revisions);

    /// <summary>
    /// Builds a configuration response from its revision and archived values.
    /// </summary>

    private async Task<ConfigurationRevisionResponse> ResponseAsync(TenantOperation op,
ConfigurationRevision revision, CancellationToken ct)
        => new(revision.Id, revision.ProductVersionId, revision.RevisionNumber,
            await reader.ReadAsync(op.Tenant.ArchivePath, revision.FileArtifact, false,
ct), revision.CreatedByUserId, revision.CreatedAt);
    /// <summary>
    /// Checks the actor's permission to access the requested configuration scope.
    /// </summary>

```



```

        private static async Task<Guid> AuthorizeAsync(TenantOperation op, ConfigurationScope
scope, bool write, CancellationToken ct)
        {
            string permission = write ? "configuration.manage" : "configuration.read";
            if (scope.InstallationId is Guid installationId)
            {
                if (scope.ProductId is not null || scope.CustomerId is not null) { throw
new ApiAccessException(404); }
                IQueryable<Installation> query = write ?
op.Access.OperationalInstallations(op.Database, permission) :
op.Access.Installations(op.Database, permission);
                return await query.Where(x => x.Id == installationId).Select(x =>
(Guid?)x.ProductId).SingleOrDefaultAsync(ct) ?? throw new ApiAccessException(404);
            }
            if (scope.ProductId is not Guid productId || !await
op.Database.Products.AnyAsync(x => x.Id == productId, ct)) { throw new ApiAccessException(404); }
            if (scope.CustomerId is Guid customerId)
            {
                IQueryable<Customer> customers = op.Access.Customers(op.Database,
permission);
                if (write) { customers = customers.Where(x => x.Status == "Active"); }
                if (!await customers.AnyAsync(x => x.Id == customerId, ct)) { throw new
ApiAccessException(404); }
            }
            else if (write && !RoleGrantPolicy.VendorAdministrator(op.Access)) { throw new
ApiAccessException(403); }
            return productId;
        }
    }
}

```

ConfigurationScope.cs

```

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Represents configuration scope data.
/// </summary>
public sealed record ConfigurationScope(Guid? ProductId = null, Guid? CustomerId = null, Guid?
InstallationId = null);

```

CustomerRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.UserAdministrationRequests;
using BridgeControl.Mappings.Shared;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

```

```

/// <summary>
/// Implements data access operations for customer.
/// </summary>
public class CustomerRepository(TenantOperationFactory operations, TenantPasswordService
credentials,
    VendorSecurityAudit audit, TimeProvider clock) : ICustomerRepository
{
    private static readonly Expression<Func<Customer, CustomerResponse>> Response = customer
=> new CustomerResponse
    { Id = customer.Id, Name = customer.Name, Status = customer.Status, CreatedAt =
customer.CreatedAt };

    /// <summary>
    /// Returns a filtered page of customer records.
    /// </summary>
    public async Task<PagedResponse<CustomerResponse>> PageAsync(TenantActor actor,
PaginationQuery page, CustomerQuery filter, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"customers.read", false, ct);
        IQueryable<Customer> query = operation.Access.Customers(operation.Database);
        if (!string.IsNullOrEmpty(filter.Name)) { string name = filter.Name.Trim();
query = query.Where(x => x.Name.Contains(name)); }
        if (!string.IsNullOrEmpty(filter.Status)) { query = query.Where(x => x.Status
== filter.Status); }
        return await new Repository<Customer, CustomerResponse>(query,
Response).PageAsync(page, x => x.Id, ct);
    }

    /// <summary>
    /// Returns the requested customer record when available.
    /// </summary>
    public async Task<CustomerResponse?> GetAsync(TenantActor actor, Guid id,
CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"customers.read", false, ct);
        return await new Repository<Customer,
CustomerResponse>(operation.Access.Customers(operation.Database).Where(x => x.Id == id),
Response).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Creates a customer record from the validated request.
    /// </summary>
    public async Task<CustomerResponse> CreateAsync(TenantActor actor, CreateCustomerRequest
request, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"customers.manage", true, ct);
        if (!RoleGrantPolicy.VendorAdministrator(operation.Access)) { throw new
ApiAccessException(403); }
        Customer customer = new() { Id = Guid.NewGuid(), Status = "Active", CreatedAt =
clock.GetUtcNow() };
        customer.MapDto(request);
        customer.Name = customer.Name.Trim();
        operation.Database.Customers.Add(customer);
        audit.Record(operation.Database, actor.UserId, "CustomerCreated", customer.Id,
entityType: "Customer");
        await operation.Database.SaveChangesAsync(ct);
        return await operation.Database.Customers.Where(x => x.Id ==
customer.Id).Select(Response).SingleAsync(ct);
    }
}

```

```

    }

    /// <summary>
    /// Updates the supplied fields of the selected customer record.
    /// </summary>
    public async Task<CustomerResponse?> UpdateAsync(TenantActor actor, Guid id,
UpdateCustomerRequest request, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"customers.manage", true, ct);
        VendorDbContext db = operation.Database;
        Customer? customer = await operation.Access.Customers(db,
"customers.manage").SingleOrDefaultAsync(x => x.Id == id, ct);
        if (customer is null) { return null; }
        if (request.NameProvided && customer.Name != request.Name)
        {
            customer.Name = request.Name!;
            audit.Record(db, actor.UserId, "CustomerRenamed", id, entityType:
"Customer");
            await db.SaveChangesAsync(ct);
        }
        return await db.Customers.Where(x => x.Id == id).Select(Response).SingleAsync(ct);
    }

    /// <summary>
    /// Changes the active status of the selected customer record.
    /// </summary>
    public async Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active,
CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"customers.manage", true, ct);
        VendorDbContext db = operation.Database;
        Customer? customer = await operation.Access.Customers(db,
"customers.manage").SingleOrDefaultAsync(x => x.Id == id, ct);
        if (customer is null) { return false; }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        string desired = active ? "Active" : "Disabled";
        bool changed = customer.Status != desired;
        customer.Status = desired;
        customer.DeactivatedAt = active ? null : customer.DeactivatedAt ??
clock.UtcNow();
        if (!active)
        {
            DateTimeOffset now = clock.UtcNow();
            await db.AgentCredentials.Where(x => x.Agent.CustomerId == id &&
x.RevokedAt == null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, now), ct);
            await db.AgentEnrollments.Where(x => x.CustomerId == id && x.UsedAt == null
&& x.RevokedAt == null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, now), ct);
            foreach (Guid userId in await db.UserAccounts.Where(x => x.CustomerId ==
id).Select(x => x.Id).ToArrayAsync(ct))
            { await credentials.RevokeCredentialsAsync(db, userId, ct, actor.UserId); }
        }
        audit.Record(db, actor.UserId, active ? "CustomerReactivated" :
"CustomerDeactivated", id, changed ? "Succeeded" : "Unchanged", "Customer");
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return true;
    }
}

```

DeploymentRepository.cs

```
using System.Linq.Expressions;
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor.DeploymentContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for deployment.
/// </summary>
public sealed class DeploymentRepository(TenantOperationFactory operations, PlatformDbContext
platform, ITenantResolver resolver,
    AgentIdentityService identity, InstallationStatePolicy policy, VendorSecurityAudit audit,
    TimeProvider clock) : IDeploymentRepository
{
    private static readonly Expression<Func<Deployment, DeploymentResponse>> Projection = x =>
new(x.Id, x.InstallationId,
    x.TargetProductVersionId, x.TargetConfigurationRevisionId, x.OperationType,
    x.Status, x.RequestedAt, x.CompletedAt);

    /// <summary>
    /// Maps stored entity data to the public response.
    /// </summary>
    private static DeploymentResponse Response(Deployment x) => new(x.Id, x.InstallationId,
x.TargetProductVersionId,
    x.TargetConfigurationRevisionId, x.OperationType, x.Status, x.RequestedAt,
    x.CompletedAt);

    /// <summary>
    /// Maps stored entity data to the public response.
    /// </summary>
    private static DeploymentAttemptResponse Response(DeploymentAttempt x) => new(x.Id,
x.AttemptNumber, x.Status, x.ErrorCode, x.ErrorMessage, x.StartedAt, x.CompletedAt);

    /// <summary>
    /// Creates a conflict error for an invalid deployment transition.
    /// </summary>
    private static ApiConflictException Conflict() => new("deployment-transition", "The
deployment cannot accept this operation in its current state.");

    /// <summary>
    /// Returns deployment history filtered to the actor's permitted installation scope.
    /// </summary>
    public async Task<PagedResponse<DeploymentResponse>> SearchAsync(TenantActor actor,
PaginationQuery page, Guid? installationId, Guid? customerId, string? status, CancellationToken
ct)
```

```

        {
            await using TenantOperation op = await operations.OpenAsync(actor,
"deployments.read", false, ct);
            IQueryable<Installation> installations = op.Access.Installations(op.Database,
"deployments.read");
            if (installationId is not null && !await installations.AnyAsync(x => x.Id ==
installationId, ct)) { throw new ApiAccessException(404); }
            if (installationId is not null) { installations = installations.Where(x => x.Id ==
installationId); }
            if (customerId is not null) { installations = installations.Where(x =>
x.CustomerId == customerId); }
            IQueryable<Deployment> query = op.Database.Deployments.Where(x =>
installations.Select(i => i.Id).Contains(x.InstallationId));
            if (!string.IsNullOrWhiteSpace(status)) { query = query.Where(x => x.Status ==
status); }
            return await new Repository<Deployment, DeploymentResponse>(query,
Projection).PageAsync(page, x => x.Id, ct);
        }

        /// <summary>
        /// Reads a deployment and its attempt history within the actor's scope.
        /// </summary>
        public async Task<DeploymentDetailResponse?> DetailAsync(TenantActor actor, Guid id,
CancellationToken ct)
        {
            await using TenantOperation op = await operations.OpenAsync(actor,
"deployments.read", false, ct);
            IQueryable<Guid> ids = op.Access.Installations(op.Database,
"deployments.read").Select(x => x.Id);
            DeploymentResponse? d = await new Repository<Deployment,
DeploymentResponse>(op.Database.Deployments.Where(x => x.Id == id &&
ids.Contains(x.InstallationId)), Projection).SingleOrDefaultAsync(ct);
            if (d is null) { return null; }
            DeploymentAttemptResponse[] attempts = await
op.Database.DeploymentAttempts.AsNoTracking().Where(x => x.DeploymentId == id).OrderBy(x =>
x.AttemptNumber)
                .Select(x => new DeploymentAttemptResponse(x.Id, x.AttemptNumber, x.Status,
x.ErrorCode, x.ErrorMessage, x.StartedAt, x.CompletedAt)).ToArrayAsync(ct);
            return new(d.Id, d.InstallationId, d.TargetProductVersionId,
d.TargetConfigurationRevisionId, d.OperationType, d.Status, d.RequestedAt, d.CompletedAt,
attempts);
        }

        /// <summary>
        /// Cancels pending deployment work and records the cancellation audit event.
        /// </summary>
        public async Task<DeploymentResponse> CancelAsync(TenantActor actor, Guid id,
CancellationToken ct)
        {
            await using TenantOperation op = await operations.OpenAsync(actor,
"deployments.manage", true, ct);
            await using IDbContextTransaction tx = await
op.Database.Database.BeginTransactionAsync(ct);
            IQueryable<Guid> ids = op.Access.Installations(op.Database,
"deployments.manage").Select(x => x.Id);
            Deployment d = await op.Database.Deployments.SingleOrDefaultAsync(x => x.Id == id
&& ids.Contains(x.InstallationId), ct) ?? throw new ApiAccessException(404);
            if (d.Status == "Cancelled") { return Response(d); }
            if (!d.IsActive || d.Status != "Pending" || await
op.Database.DeploymentAttempts.AnyAsync(x => x.DeploymentId == id && x.CompletedAt == null, ct))
            { throw Conflict(); }
            d.Status = "Cancelled"; d.IsActive = false; d.CompletedAt = clock.UtcNow();

```

```

        audit.Record(op.Database, actor.UserId, "DeploymentCancelledByUser", id,
entityType: "Deployment");
        await op.Database.SaveChangesAsync(ct); await tx.CommitAsync(ct); return
Response(d);
    }

    /// <summary>
    /// Returns a page of eligible pending work for the authenticated agent.
    /// </summary>
    public async Task<AgentDeploymentPage> PendingAsync(AgentActor actor, PaginationQuery
page, string status, CancellationToken ct)
    {
        if (!string.Equals(status, "pending", StringComparison.OrdinalIgnoreCase)) { throw
Conflict(); }
        await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        if (await identity.CurrentAsync(tenant, actor, ct) is null) { throw new
ApiAccessException(401); }
        VendorDbContext db = tenant.Database;
        IQueryable<Guid> owned = db.Installations.Where(x => x.AgentId ==
actor.AgentId).Select(x => x.Id);
        Deployment[] candidates = await db.Deployments.Where(x =>
owned.Contains(x.InstallationId) && x.IsActive && x.Status == "Pending")
.OrderBy(x => x.RequestedAt).ThenBy(x =>
x.Id).Skip(page.Offset).Take(page.PageSize + 1).ToArrayAsync(ct);
        List<DeploymentResponse> result = [];
        foreach (Deployment? d in candidates.Take(page.PageSize))
        {
            try { await EligibleAsync(tenant, d, ct); result.Add(Response(d)); }
            catch (Exception e) when (e is ApiConflictException or
ConfigurationValidationException or ApiAccessException) { continue; }
        }
        return new(result, candidates.Length > page.PageSize);
    }

    /// <summary>
    /// Atomically starts or recovers an attempt for the agent's deployment.
    /// </summary>
    public async Task<StartDeploymentAttemptResponse> StartAsync(AgentActor actor, Guid id,
CancellationToken ct)
    {
        await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        await using IDbContextTransaction tx = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Deployment d = await OwnedAsync(tenant, actor, id, ct);
        DeploymentAttempt? last = await db.DeploymentAttempts.Where(x => x.DeploymentId ==
id).OrderByDescending(x => x.AttemptNumber).FirstOrDefaultAsync(ct);
        if (last is not null && last.CompletedAt == null && last.AgentId == actor.AgentId)
        {
            Installation installation = await db.Installations.SingleAsync(x => x.Id ==
d.InstallationId, ct);
            if (!d.IsActive || d.Status != "Running" ||
installation.DesiredStateRevisionId != d.DesiredStateRevisionId) { throw Conflict(); }
            await policy.RequireEligibleAsync(db, installation.Id, ct);
            await policy.TargetAsync(tenant, installation, d.TargetProductVersionId,
d.TargetConfigurationRevisionId, ct);

```

```

        return new(last.Id, last.AttemptNumber);
    }
    await EligibleAsync(tenant, d, ct);
    DeploymentAttempt attempt = new DeploymentAttempt { DeploymentId = id,
InstallationId = d.InstallationId, AgentId = actor.AgentId, AttemptNumber = (last?.AttemptNumber
?? 0) + 1, StartedAt = clock.UtcNow() };
    db.DeploymentAttempts.Add(attempt); d.Status = "Running";
    audit.Record(db, actor.AgentId, "DeploymentAttemptStarted", id, actorType:
"Agent", entityType: "Deployment", resourceId: attempt.Id);
    await db.SaveChangesAsync(ct); await tx.CommitAsync(ct); return new(attempt.Id,
attempt.AttemptNumber);
    }

    /// <summary>
    /// Records an ordered deployment-attempt report and its terminal outcome.
    /// </summary>
    public async Task<DeploymentAttemptResponse> UpdateAsync(AgentActor actor, Guid id, Guid
attemptId, UpdateDeploymentAttemptRequest request, CancellationToken ct)
    {
        await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        await using IDbContextTransaction tx = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Deployment d = await OwnedAsync(tenant, actor, id, ct);
        DeploymentAttempt a = await db.DeploymentAttempts.SingleOrDefaultAsync(x => x.Id
== attemptId && x.DeploymentId == id && x.AgentId == actor.AgentId, ct) ?? throw new
ApiAccessException(404);
        string fingerprint =
Convert.ToHexString(SHA256.HashData(Encoding.UTF8.GetBytes(JsonSerializer.Serialize(request))));
        if (a.LastReportHash == fingerprint) { return Response(a); }
        bool terminal = request.Status is "Succeeded" or "Failed" or "RestartRequired";
        int Rank(string s) => s switch { "Running" => 0, "Installing" => 1, "Configuring"
=> 2, "Succeeded" or "Failed" or "RestartRequired" => 3, _ => -1 };
        if (a.CompletedAt is not null || Rank(request.Status) < Rank(a.Status) ||
Rank(request.Status) < 0
            || (d.OperationType == "Configure" && request.Status == "Installing")
            || terminal != request.CompletedAt.HasValue || request.CompletedAt <
a.StartedAt || request.CompletedAt > clock.UtcNow().AddMinutes(5)
            || (!terminal && (request.ErrorCode is not null || request.ErrorMessage is
not null))
            || (request.Status == "Succeeded" && (request.ErrorCode is not null ||
request.ErrorMessage is not null))) { throw Conflict(); }
        a.Status = request.Status; a.CompletedAt = request.CompletedAt; a.LastReportHash =
fingerprint;
        if (terminal) { a.ResultReceivedAt = clock.UtcNow(); }
        a.ErrorCode = request.Status is "Failed" or "RestartRequired" ?
SafeCode(request.ErrorCode, request.Status) : null;
        a.ErrorMessage = a.ErrorCode is null ? null : SafeMessage(a.ErrorCode);
        if (terminal && d.IsActive)
        {
            d.Status = request.Status == "Failed" ? (a.AttemptNumber >= 3 ? "Failed" :
"Pending") : "AwaitingState";
            if (d.Status == "Failed") { d.IsActive = false; d.CompletedAt =
clock.UtcNow(); }
        }
        audit.Record(db, actor.AgentId, "DeploymentAttempt" + request.Status, id,
actorType: "Agent", entityType: "Deployment", resourceId: a.Id);
        await db.SaveChangesAsync(ct); await tx.CommitAsync(ct); return Response(a);
    }
}

```

```

    /// <summary>
    /// Loads deployment work owned by the authenticated agent.
    /// </summary>
    private async Task<Deployment> OwnedAsync(TenantContext tenant, AgentActor actor, Guid id,
CancellationToken ct)
    {
        if (await identity.CurrentAsync(tenant, actor, ct) is null) { throw new
ApiAccessException(401); }
        IQueryable<Guid> ids = tenant.Database.Installations.Where(x => x.AgentId ==
actor.AgentId).Select(x => x.Id);
        return await tenant.Database.Deployments.SingleOrDefaultAsync(x => x.Id == id &&
ids.Contains(x.InstallationId), ct) ?? throw new ApiAccessException(404);
    }

    /// <summary>
    /// Checks that the deployment target remains eligible for dispatch.
    /// </summary>
    private async Task EligibleAsync(TenantContext tenant, Deployment d, CancellationTokens ct)
    {
        VendorDbContext db = tenant.Database;
        Installation i = await db.Installations.SingleAsync(x => x.Id == d.InstallationId,
ct);

        if (!d.IsActive || d.Status != "Pending" || i.DesiredStateRevisionId !=
d.DesiredStateRevisionId
            || await db.DeploymentAttempts.AnyAsync(x => x.InstallationId == i.Id &&
x.CompletedAt == null, ct)) { throw Conflict(); }
        DeploymentAttempt? last = await db.DeploymentAttempts.Where(x => x.DeploymentId ==
d.Id).OrderByDescending(x => x.AttemptNumber).FirstOrDefaultAsync(ct);
        if (last is not null && (last.AttemptNumber >= 3 || last.Status != "Failed" ||
last.ResultReceivedAt is null || last.ResultReceivedAt.Value.AddSeconds(30 * last.AttemptNumber)
> clock.GetUtcNow())) { throw Conflict(); }
        await policy.RequireEligibleAsync(db, i.Id, ct);
        StateTarget target = await policy.TargetAsync(tenant, i, d.TargetProductVersionId,
d.TargetConfigurationRevisionId, ct);
        InstallationStateReport? report = await
db.InstallationStateReports.SingleOrDefaultAsync(x => x.Id == i.ReportedStateReportId &&
x.AgentId == i.AgentId, ct);
        string classification = StateReconciliationProcessor.Classify(report, target,
clock.GetUtcNow());
        if (classification is "Matching" or "Unknown" || (last is null && classification
!= d.OperationType)
            || (d.OperationType == "Configure" && classification != "Configure")) {
throw Conflict(); }
    }

    /// <summary>
    /// Returns a bounded, safe deployment result code.
    /// </summary>
    private static string SafeCode(string? code, string status) => status == "RestartRequired"
? "RESTART_REQUIRED" : code switch
    {
        "MSI_INSTALL_FAILED" or "CONFIGURATION_APPLY_FAILED" or "DOWNLOAD_FAILED" or
"INTERRUPTED" => code,
        _ => "UNKNOWN_ERROR"
    };

    /// <summary>
    /// Returns a deployment message suitable for public operational history.
    /// </summary>
    private static string SafeMessage(string code) => code switch
    {

```



```

        "MSI_INSTALL_FAILED" => "The installer failed.",
        "CONFIGURATION_APPLY_FAILED" => "Configuration application failed.",
        "DOWNLOAD_FAILED" => "Artifact transfer or integrity verification failed.",
        "INTERRUPTED" => "Execution was interrupted.",
        "RESTART_REQUIRED" => "A machine restart is required.",
        _ => "The Agent reported a deployment failure."
    };
}

```

EffectiveConfigurationRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for effective configuration.
/// </summary>
public sealed class EffectiveConfigurationRepository(TenantOperationFactory operations,
    ConfigurationArtifactReader reader) : IEffectiveConfigurationRepository
{
    /// <summary>
    /// Represents read model data.
    /// </summary>
    private sealed record ReadModel(Guid Id, Guid InstallationId, Guid VersionId, int Number,
        DateTimeOffset CreatedAt, FileArtifact Artifact, List<Guid> Sources);

    /// <summary>
    /// Projects an effective configuration revision into a read model with its file artifact
    and ordered source revision IDs.
    /// </summary>
    private static readonly Expression<Func<EffectiveConfigurationRevision, ReadModel>>
    Projection = x
        => new(x.Id, x.InstallationId, x.ProductVersionId, x.RevisionNumber, x.CreatedAt,
            x.FileArtifact, x.SourceRevisions.OrderBy(s => s.ConfigurationRevisionId).Select(s =>
                s.ConfigurationRevisionId).ToList());

    /// <summary>
    /// Builds the effective configuration query within the caller's permitted scope.
    /// </summary>
    private static IQueryable<EffectiveConfigurationRevision> Scoped(TenantOperation op, Guid
        installationId) => op.Database.EffectiveConfigurationRevisions.Where(x => x.InstallationId ==
            installationId
                && op.Access.Installations(op.Database, "configuration.read").Any(i => i.Id ==
                    x.InstallationId));

    /// <summary>
    /// Returns the current effective configuration revision.
    /// </summary>
    public async Task<EffectiveConfigurationResponse?> CurrentAsync(TenantActor actor, Guid
        installationId, CancellationToken ct)
    {

```

```

        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        IQueryable<EffectiveConfigurationRevision> query = Scoped(op,
installationId).Where(x => x.Installation.CurrentEffectiveConfigurationRevisionId == x.Id);
        ReadModel? item = await new Repository<EffectiveConfigurationRevision,
ReadModel>(query, Projection).SingleOrDefaultAsync(ct);
        return item is null ? null : await ResponseAsync(op, item, ct);
    }

    /// <summary>
    /// Returns the requested effective configuration record when available.
    /// </summary>
    public async Task<EffectiveConfigurationResponse?> GetAsync(TenantActor actor, Guid
installationId, Guid revisionId, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        ReadModel? item = await new Repository<EffectiveConfigurationRevision,
ReadModel>(Scoped(op, installationId).Where(x => x.Id == revisionId),
Projection).SingleOrDefaultAsync(ct);
        return item is null ? null : await ResponseAsync(op, item, ct);
    }

    /// <summary>
    /// Returns a page of historical effective configuration revisions.
    /// </summary>
    public async Task<PagedResponse<EffectiveConfigurationResponse>> HistoryAsync(TenantActor
actor, Guid installationId, PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.read", false, ct);
        if (!await op.Access.Installations(op.Database, "configuration.read").AnyAsync(x
=> x.Id == installationId, ct)) { throw new ApiAccessException(404); }
        PagedResponse<ReadModel> items = await new
Repository<EffectiveConfigurationRevision, ReadModel>(Scoped(op, installationId),
Projection).PageAsync(page, x => x.RevisionNumber, ct);
        return await items.MapAsync((item, token) => ResponseAsync(op, item, token), ct);
    }

    /// <summary>
    /// Builds the effective-configuration response with verified archived values and source
lineage.
    /// </summary>
    private async Task<EffectiveConfigurationResponse> ResponseAsync(TenantOperation op,
ReadModel item, CancellationToken ct)
        => new(item.Id, item.InstallationId, item.VersionId, item.Number, await
reader.ReadAsync(op.Tenant.ArchivePath, item.Artifact, false, ct), item.Sources, item.CreatedAt);
}

```

IAccessAssignmentRepository.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.UserAdministrationRequests;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for access assignment.

```

```

/// </summary>
public interface IAccessAssignmentRepository
{
    /// <summary>
    /// Lists the roles assigned to the selected user.
    /// </summary>
    Task<PagedResponse<RoleDto>> RolesAsync(TenantActor actor, PaginationQuery page,
CancellationToken ct);

    /// <summary>
    /// Lists the permissions available through the selected user's roles.
    /// </summary>
    Task<PagedResponse<PermissionDto>> PermissionsAsync(TenantActor actor, PaginationQuery
page, CancellationToken ct);

    /// <summary>
    /// Replaces the user's role assignments after checking the actor's grant authority.
    /// </summary>
    Task ReplaceRolesAsync(TenantActor actor, Guid userId, SetUserRolesRequest request,
CancellationToken ct);

    /// <summary>
    /// Lists the customer's access assignments for the selected technician.
    /// </summary>
    Task<SetCustomerAccessRequest> CustomersAsync(TenantActor actor, Guid userId,
CancellationToken ct);

    /// <summary>
    /// Replaces the customer's access assignments for the selected technician.
    /// </summary>
    Task ReplaceCustomersAsync(TenantActor actor, Guid userId, SetCustomerAccessRequest
request, CancellationToken ct);

    /// <summary>
    /// Lists installation access assignments for the selected user.
    /// </summary>
    Task<SetInstallationAccessRequest> InstallationsAsync(TenantActor actor, Guid userId,
CancellationToken ct);

    /// <summary>
    /// Replaces installation access assignments for the selected user.
    /// </summary>
    Task ReplaceInstallationsAsync(TenantActor actor, Guid userId,
SetInstallationAccessRequest request, CancellationToken ct);
}

```

IConfigurationMigrationExecutionRepository.cs

```

using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.ConfigurationMigrationExecution;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for configuration migration execution.
/// </summary>
public interface IConfigurationMigrationExecutionRepository
{
    /// <summary>

```

```

        /// Creates a configuration migration preview for the authorized installation.
        /// </summary>
        Task<ConfigurationMigrationPreviewResponse> PreviewAsync(TenantActor actor, Guid
installationId, ConfigurationMigrationPreviewRequest request, CancellationToken ct);

        /// <summary>
        /// Applies an approved configuration migration for the authorized installation.
        /// </summary>
        Task<EffectiveConfigurationResponse> ApplyAsync(TenantActor actor, Guid installationId,
ApplyConfigurationMigrationRequest request, CancellationToken ct);
    }

```

IConfigurationMigrationRepository.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for configuration migration.
/// </summary>
public interface IConfigurationMigrationRepository
{
    /// <summary>
    /// Returns the matching configuration migration records.
    /// </summary>
    Task<PagedResponse<ConfigurationMigrationDto>> ListAsync(TenantActor actor, Guid product,
PaginationQuery page, CancellationToken ct);

    /// <summary>
    /// Returns the requested configuration migration record when available.
    /// </summary>
    Task<ConfigurationMigrationDto?> GetAsync(TenantActor actor, Guid product, Guid id,
CancellationToken ct);

    /// <summary>
    /// Validates and persists the configuration migration definition.
    /// </summary>
    Task<ConfigurationMigrationDto> SaveAsync(TenantActor actor, Guid product, Guid? id,
CreateConfigurationMigrationRequest request, CancellationToken ct);
}

```

IConfigurationRepository.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for configuration.
/// </summary>
public interface IConfigurationRepository
{

```

```

    /// <summary>
    /// Returns the current configuration revision.
    /// </summary>
    Task<ConfigurationRevisionResponse?> CurrentAsync(TenantActor actor, ConfigurationScope
scope, CancellationToken ct);

    /// <summary>
    /// Returns a page of historical configuration revisions.
    /// </summary>
    Task<PagedResponse<ConfigurationRevisionResponse>> HistoryAsync(TenantActor actor,
ConfigurationScope scope, PaginationQuery page, CancellationToken ct);

    /// <summary>
    /// Writes a validated configuration revision at the requested scope.
    /// </summary>
    Task<ConfigurationRevisionResponse> SetAsync(TenantActor actor, ConfigurationScope scope,
UpdateConfigurationRequest request, CancellationToken ct);
}

```

ICustomerRepository.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.UserAdministrationRequests;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for customer.
/// </summary>
public interface ICustomerRepository
{
    /// <summary>
    /// Returns a filtered page of customer records.
    /// </summary>
    Task<PagedResponse<CustomerResponse>> PageAsync(TenantActor actor, PaginationQuery page,
CustomerQuery filter, CancellationToken ct);

    /// <summary>
    /// Returns the requested customer record when available.
    /// </summary>
    Task<CustomerResponse?> GetAsync(TenantActor actor, Guid id, CancellationToken ct);

    /// <summary>
    /// Creates a customer record from the validated request.
    /// </summary>
    Task<CustomerResponse> CreateAsync(TenantActor actor, CreateCustomerRequest request,
CancellationToken ct);

    /// <summary>
    /// Updates the supplied fields of the selected customer record.
    /// </summary>
    Task<CustomerResponse?> UpdateAsync(TenantActor actor, Guid id, UpdateCustomerRequest
request, CancellationToken ct);

    /// <summary>
    /// Changes the active status of the selected customer record.
    /// </summary>
    Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active, CancellationToken ct);
}

```

```
}
```

IDeploymentRepository.cs

```
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor.DeploymentContracts;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Represents agent deployment page data.
/// </summary>
public sealed record AgentDeploymentPage(IReadOnlyList<DeploymentResponse> Items, bool HasMore);
/// <summary>
/// Defines data access operations for deployment.
/// </summary>
public interface IDeploymentRepository
{
    /// <summary>
    /// Returns deployment history filtered to the actor's permitted installation scope.
    /// </summary>
    Task<PagedResponse<DeploymentResponse>> SearchAsync(TenantActor actor, PaginationQuery
page, Guid? installationId, Guid? customerId, string? status, CancellationToken ct);

    /// <summary>
    /// Reads a deployment and its attempt history within the actor's scope.
    /// </summary>
    Task<DeploymentDetailResponse?> DetailAsync(TenantActor actor, Guid id, CancellationToken
ct);

    /// <summary>
    /// Cancels pending deployment work and records the cancellation audit event.
    /// </summary>
    Task<DeploymentResponse> CancelAsync(TenantActor actor, Guid id, CancellationToken ct);

    /// <summary>
    /// Returns a page of eligible pending work for the authenticated agent.
    /// </summary>
    Task<AgentDeploymentPage> PendingAsync(AgentActor actor, PaginationQuery page, string
status, CancellationToken ct);

    /// <summary>
    /// Atomically starts or recovers an attempt for the agent's deployment.
    /// </summary>
    Task<StartDeploymentAttemptResponse> StartAsync(AgentActor actor, Guid id,
CancellationToken ct);

    /// <summary>
    /// Records an ordered deployment-attempt report and its terminal outcome.
    /// </summary>
    Task<DeploymentAttemptResponse> UpdateAsync(AgentActor actor, Guid id, Guid attemptId,
UpdateDeploymentAttemptRequest request, CancellationToken ct);
}
```

IEffectiveConfigurationRepository.cs

```

using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for effective configuration.
/// </summary>
public interface IEffectiveConfigurationRepository
{
    /// <summary>
    /// Returns the current effective configuration revision.
    /// </summary>
    Task<EffectiveConfigurationResponse?> CurrentAsync(TenantActor actor, Guid installationId,
CancellationTokens ct);

    /// <summary>
    /// Returns a page of historical effective configuration revisions.
    /// </summary>
    Task<PagedResponse<EffectiveConfigurationResponse>> HistoryAsync(TenantActor actor, Guid
installationId, PaginationQuery page, CancellationTokens ct);

    /// <summary>
    /// Returns the requested effective configuration record when available.
    /// </summary>
    Task<EffectiveConfigurationResponse?> GetAsync(TenantActor actor, Guid installationId,
Guid revisionId, CancellationTokens ct);
}

```

InstallationRepository.cs

```

using BridgeControl.DTOs.AgentDTOs.AgentContracts;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.DTOs.Vendor.LicenseContracts;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for installation.
/// </summary>
public interface IInstallationRepository
{
    /// <summary>
    /// Creates a installation record from the validated request.
    /// </summary>
    Task<InstallationResponse> CreateAsync(TenantActor actor, AssignProductRequest request,
CancellationTokens ct);

    /// <summary>
    /// Returns a filtered page of installation records.
    /// </summary>
    Task<PagedResponse<InstallationResponse>> PageAsync(TenantActor actor, PaginationQuery
page, InstallationQuery filter, CancellationTokens ct);

    /// <summary>
    /// Returns the requested installation record when available.

```

```

    /// </summary>
    Task<InstallationResponse?> GetAsync(TenantActor actor, Guid id, CancellationToken ct);

    /// <summary>
    /// Updates the supplied fields of the selected installation record.
    /// </summary>
    Task<InstallationResponse?> UpdateAsync(TenantActor actor, Guid id,
UpdateInstallationRequest request, CancellationToken ct);

    /// <summary>
    /// Changes the active status of the selected installation record.
    /// </summary>
    Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active, CancellationToken ct);
}

```

InstallationStateRepository.cs

```

using BridgeControl.DTOs.Vendor.InstallationStateContracts;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;
/// <summary>
/// Defines data access operations for installation state.
/// </summary>
public interface IInstallationStateRepository
{
    /// <summary>
    /// Reads the desired state of an installation within the actor's permitted scope.
    /// </summary>
    Task<DesiredStateResponse?> DesiredAsync(TenantActor actor, Guid installation,
CancellationToken ct);

    /// <summary>
    /// Reads the latest reported state of an authorized installation.
    /// </summary>
    Task<ReportedStateResponse?> ReportedAsync(TenantActor actor, Guid installation,
CancellationToken ct);

    /// <summary>
    /// Creates an eligible desired-state revision and schedules reconciliation atomically.
    /// </summary>
    Task<DesiredStateResponse> SetDesiredAsync(TenantActor actor, Guid installation,
SetDesiredStateRequest request, CancellationToken ct);

    /// <summary>
    /// Returns the authenticated agent's desired target and pinned artifact links.
    /// </summary>
    Task<AgentDesiredState?> AgentDesiredAsync(AgentActor actor, Guid installation,
CancellationToken ct);

    /// <summary>
    /// Records ordered, deduplicated installation evidence and schedules reconciliation.
    /// </summary>
    Task<ReportedStateResponse> ReportAsync(AgentActor actor, Guid installation,
ReportInstallationStateRequest request, CancellationToken ct);

    /// <summary>
    /// Opens a target artifact after checking agent ownership and current entitlement.
    /// </summary>

```



```

        Task<AgentArtifactContent> ArtifactAsync(AgentActor actor, Guid artifact,
CancellationTokens ct);
    }

```

ILicenseRepository.cs

```

using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.LicenseContracts;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for license.
/// </summary>
public interface ILicenseRepository
{
    /// <summary>
    /// Returns a filtered page of license records.
    /// </summary>
    Task<PagedResponse<LicenseResponse>> PageAsync(TenantActor actor, PaginationQuery page,
InstallationQuery filter, CancellationTokens ct);

    /// <summary>
    /// Returns the requested license record when available.
    /// </summary>
    Task<LicenseResponse?> GetAsync(TenantActor actor, Guid id, CancellationTokens ct);

    /// <summary>
    /// Creates a license record from the validated request.
    /// </summary>
    Task<LicenseResponse> CreateAsync(TenantActor actor, CreateLicenseRequest request,
CancellationTokens ct);

    /// <summary>
    /// Updates the supplied fields of the selected license record.
    /// </summary>
    Task<LicenseResponse?> UpdateAsync(TenantActor actor, Guid id, UpdateLicenseRequest
request, CancellationTokens ct);

    /// <summary>
    /// Changes the active status of the selected license record.
    /// </summary>
    Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active, CancellationTokens ct);

    /// <summary>
    /// Assigns or removes an installation license while enforcing ownership and capacity.
    /// </summary>
    Task<bool> AssignAsync(TenantActor actor, Guid installationId, Guid? licenseId,
CancellationTokens ct);
}

```

InstallationRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;

```

```

using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.DeploymentSettings;
using BridgeControl.DTOS.Vendor.LicenseContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Repositories.Vendor;
/// <summary>
/// Implements data access operations for installation.
/// </summary>
public sealed class InstallationRepository(TenantOperationFactory operations, LicenseEligibility
eligibility, VendorSecurityAudit audit) : IInstallationRepository
{
    /// <summary>
    /// Creates a installation record from the validated request.
    /// </summary>
    public async Task<InstallationResponse> CreateAsync(TenantActor actor,
AssignProductRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"installations.manage", true, ct);
        VendorDbContext db = op.Database;
        // Customer users with individual installation grants cannot provision new
machines.
        if (!op.Access.Roles.Any(x => x is "VendorAdministrator" or "VendorTechnician" or
"CustomerAdministrator")) { throw new ApiAccessException(403); }
        IQueryable<Guid> customers = op.Access.Customers(db,
"installations.manage").Where(x => x.Status == "Active").Select(x => x.Id);
        Agent agent = await db.Agents.SingleOrDefaultAsync(x => x.Id == request.AgentId &&
x.Status == "Active" && customers.Contains(x.CustomerId), ct) ?? throw new
ApiAccessException(404);
        if (!await db.Products.AnyAsync(x => x.Id == request.ProductId && x.Status ==
"Active", ct)) { throw new ApiAccessException(404); }
        List<Installation> matches = await db.Installations.Where(x => x.AgentId ==
agent.Id && x.ProductId == request.ProductId).Take(2).ToListAsync(ct);
        if (matches.Count > 0)
        {
            if (matches.Count != 1 || matches[0].Status != "Active") { throw new
ApiConflictException("installation-conflict", "An existing assignment must be reviewed or
reactivated."); }
            return await db.Installations.Where(x => x.Id ==
matches[0].Id).Select(Response).SingleAsync(ct);
        }
        if (!await db.ProductVersions.AnyAsync(x => x.ProductId == request.ProductId &&
x.Status == "Published" && x.DeploymentSettingsJson != null, ct))
        { throw new ApiConflictException("deployment-settings-required", "Configure
deployment settings for a published product version first."); }
        Installation installation = new Installation
        {
            AgentId = agent.Id,
            CustomerId = agent.CustomerId,
            ProductId = request.ProductId,
            Name = request.Name,
            RegisteredAt = DateTimeOffset.UtcNow
        };
        installation.RegistrationName = "assigned-" + installation.Id.ToString("N");
        db.Installations.Add(installation);
    }
}

```

```

        db.EffectiveConfigurationSignals.Add(new() { InstallationId = installation.Id,
CreatedAt = installation.RegisteredAt });
        audit.Record(db, actor.UserId, "ProductAssignedToAgent", installation.Id,
entityType: "Installation");
        await db.SaveChangesAsync(ct);
        return await db.Installations.Where(x => x.Id ==
installation.Id).Select(Response).SingleOrDefaultAsync(ct);
    }

    private static readonly Expression

```

```

        audit.Record(db, actor.UserId, "InstallationRenamed", id, entityType:
"Installation"); await db.SaveChangesAsync(ct);
    }
    return await db.Installations.Where(x => x.Id ==
id).Select(Response).SingleOrDefaultAsync(ct);
}

/// <summary>
/// Changes the active status of the selected installation record.
/// </summary>
public async Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active,
Cancellation.Token ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"installations.manage", true, ct);
    VendorDbContext db = operation.Database;
    Installation? installation = await operation.Access.Installations(db,
"installations.manage").Include(x => x.Customer).Include(x => x.Product).SingleOrDefaultAsync(x
=> x.Id == id, ct);
    if (installation is null) { return false; }
    string desired = active ? "Active" : "Disabled";
    if (installation.Status == desired) { return true; }
    if (active)
    {
        if (installation.Customer.Status != "Active" || installation.Product.Status
!= "Active"
            || (installation.AgentId is Guid agent && !await
db.Agents.AnyAsync(x => x.Id == agent && x.CustomerId == installation.CustomerId && x.Status ==
"Active", ct)))
        { throw new ApiConflictException("installation-unavailable", "The
installation's customer, product or Agent is disabled."); }
        License? license = await db.InstallationLicenseAssignments.Where(x =>
x.InstallationId == id && x.RemovedAt == null).Select(x => x.License).SingleOrDefaultAsync(ct);
        if (license is not null) { await eligibility.RequireAssignmentAsync(db,
installation, license, ct); }
    }
    installation.Status = desired;
    audit.Record(db, actor.UserId, active ? "InstallationReactivated" :
"InstallationDeactivated", id, entityType: "Installation");
    await db.SaveChangesAsync(ct); return true;
}
}

```

InstallationStateRepository.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Vendor.InstallationStateContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Repositories.Vendor;

```

```

/// <summary>
/// Implements data access operations for installation state.
/// </summary>
public sealed class InstallationStateRepository(TenantOperationFactory operations,
PlatformDbContext platform, ITenantResolver resolver,
    AgentIdentityService identity, InstallationStatePolicy policy,
EffectiveConfigurationGenerator generator, VendorSecurityAudit audit, TimeProvider clock) :
IInstallationStateRepository
{
    /// <summary>
    /// Reads the desired state of an installation within the actor's permitted scope.
    /// </summary>
    public async Task<DesiredStateResponse?> DesiredAsync(TenantActor actor, Guid
installation, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"installations.read", false, ct);
        IQueryable<Guid?> ids = op.Access.Installations(op.Database).Where(x => x.Id ==
installation).Select(x => x.DesiredStateRevisionId);
        return await new Repository<DesiredStateRevision,
DesiredStateResponse>(op.Database.DesiredStateRevisions.Where(x => ids.Contains(x.Id)),
            x => new(x.InstallationId, x.ProductVersionId,
x.EffectiveConfigurationRevisionId, x.UpdatedAt)).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Reads the latest reported state of an authorized installation.
    /// </summary>
    public async Task<ReportedStateResponse?> ReportedAsync(TenantActor actor, Guid
installation, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"installations.read", false, ct);
        IQueryable<Guid?> ids = op.Access.Installations(op.Database).Where(x => x.Id ==
installation).Select(x => x.ReportedStateReportId);
        return await new Repository<InstallationStateReport,
ReportedStateResponse>(op.Database.InstallationStateReports.Where(x => ids.Contains(x.Id)),
            x => new(x.InstallationId, x.InstalledProductVersionId,
x.EffectiveConfigurationRevisionId, x.ConfigurationHash, x.Status,
x.ReportedAt)).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Creates an eligible desired-state revision and schedules reconciliation atomically.
    /// </summary>
    public async Task<DesiredStateResponse> SetDesiredAsync(TenantActor actor, Guid
installationId, SetDesiredStateRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"installations.manage", true, ct); VendorDbContext db = op.Database;
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Installation installation = await
op.Access.OperationalInstallations(db).SingleOrDefaultAsync(x => x.Id == installationId, ct) ??
throw new ApiAccessException(404);
        await policy.RequireEligibleAsync(db, installationId, ct);
        await policy.TargetAsync(op.Tenant, installation, request.ProductVersionId,
request.EffectiveConfigurationRevisionId, ct);
        DesiredStateRevision? current = await
db.DesiredStateRevisions.SingleOrDefaultAsync(x => x.Id == installation.DesiredStateRevisionId,
ct);
    }
}

```

```

        if (current is not null && current.ProductVersionId == request.ProductVersionId &&
current.EffectiveConfigurationRevisionId == request.EffectiveConfigurationRevisionId)
        { await transaction.CommitAsync(ct); return Response(current); }
        // A version change must not silently discard an existing customer/installation
override layer.
        EffectiveConfigurationRevision? previous = await
db.EffectiveConfigurationRevisions.SingleOrDefaultAsync(x => x.Id ==
installation.CurrentEffectiveConfigurationRevisionId, ct);
        if (previous is not null && previous.ProductVersionId != request.ProductVersionId)
        {
            Guid[] before = await db.EffectiveConfigurationSourceRevisions.Where(x =>
x.EffectiveConfigurationRevisionId == previous.Id)
                .Select(x => x.ConfigurationRevision).Where(x =>
x.ConfigurationProfile.CustomerId != null || x.ConfigurationProfile.InstallationId !=
null).Select(x => x.ConfigurationProfileId).ToArrayAsync(ct);
            Guid[] after = await db.EffectiveConfigurationSourceRevisions.Where(x =>
x.EffectiveConfigurationRevisionId == request.EffectiveConfigurationRevisionId).Select(x =>
x.ConfigurationRevision.ConfigurationProfileId).ToArrayAsync(ct);
            if (before.Except(after).Any()) { throw new
ApiConflictException("configuration-migration-required", "Prepare or migrate the override layers
for the target version first."); }
        }
        Guid fresh = await generator.GenerateLockedAsync(op.Tenant, installationId,
request.ProductVersionId, true, ct);
        if (fresh != request.EffectiveConfigurationRevisionId) { throw new
ApiConflictException("desired-configuration-stale", "Select the latest valid configuration for
this target version."); }
        DesiredStateRevision desired = new DesiredStateRevision
        {
            InstallationId = installationId,
            ProductVersionId = request.ProductVersionId,
            EffectiveConfigurationRevisionId = fresh,
            RevisionNumber = checked(++installation.LastDesiredStateRevisionNumber),
            CreatedByUserId = actor.UserId,
            UpdatedAt = clock.GetUtcNow()
        };
        db.DesiredStateRevisions.Add(desired); await db.SaveChangesAsync(ct);
        installation.DesiredStateRevisionId = desired.Id;
        await SupersedeAsync(db, installationId, desired.Id, clock.GetUtcNow(), ct);
        await InstallationStatePolicy.EnqueueAsync(db, installationId, clock.GetUtcNow(),
ct);
        audit.Record(db, actor.UserId, "InstallationDesiredStateSet", installationId,
entityType: "Installation", resourceId: desired.Id);
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return
Response(desired);
    }

    /// <summary>
    /// Returns the authenticated agent's desired target and pinned artifact links.
    /// </summary>
    public async Task<AgentDesiredState?> AgentDesiredAsync(AgentActor actor, Guid
installationId, CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        Installation installation = await AgentInstallationAsync(tenant, actor,
installationId, true, ct);
        await policy.RequireEligibleAsync(tenant.Database, installationId, ct);
    }

```

```

        DesiredStateRevision? desired = await
tenant.Database.DesiredStateRevisions.SingleOrDefaultAsync(x => x.Id ==
installation.DesiredStateRevisionId, ct);
        if (desired is null) { return null; }
        StateTarget target = await policy.TargetAsync(tenant, installation,
desired.ProductVersionId, desired.EffectiveConfigurationRevisionId, ct);
        return new(Response(desired), target.Installer.Id,
target.Configuration.FileArtifactId);
    }

    /// <summary>
    /// Records ordered, deduplicated installation evidence and schedules reconciliation.
    /// </summary>
    public async Task<ReportedStateResponse> ReportAsync(AgentActor actor, Guid
installationId, ReportInstallationStateRequest request, CancellationToken ct)
    {
        if (request.ReportedAt == default || request.ReportedAt >
clock.UtcNow().AddMinutes(5)) { throw new AgentValidationException("ReportedAt must be a real
timestamp no more than five minutes ahead of the server."); }
        if (request.Status is not ("Running" or "Ready" or "Stopped" or "NotInstalled" or
"Unknown" or "Failed" or "RestartRequired")) { throw new AgentValidationException("Unsupported
installation status."); }
        if (request.Status == "Ready" && (request.InstalledProductVersionId is null ||
request.EffectiveConfigurationRevisionId is null || request.ConfigurationHash is null))
        { throw new AgentValidationException("Ready requires installed version and
configuration evidence."); }
        string? hash = request.ConfigurationHash?.ToLowerInvariant();
        if (hash is not null && (hash.Length != 64 || hash.Any(c => !Uri.IsHexDigit(c))))
        { throw new AgentValidationException("ConfigurationHash must be a SHA-256 hexadecimal value."); }
        if (request.Status == "NotInstalled" && (request.InstalledProductVersionId is not
null || request.EffectiveConfigurationRevisionId is not null || hash is not null))
        { throw new AgentValidationException("NotInstalled must not claim installed
version or configuration evidence."); }
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Installation installation = await AgentInstallationAsync(tenant, actor,
installationId, false, ct);
        if (request.InstalledProductVersionId is Guid version && !await
db.ProductVersions.AnyAsync(x => x.Id == version && x.ProductId == installation.ProductId, ct)) {
throw new ApiAccessException(404); }
        if (request.EffectiveConfigurationRevisionId is Guid configuration &&
(request.InstalledProductVersionId is null || !await
db.EffectiveConfigurationRevisions.AnyAsync(x => x.Id == configuration && x.InstallationId ==
installationId && x.ProductVersionId == request.InstalledProductVersionId, ct))) { throw new
ApiAccessException(404); }
        InstallationStateReport? duplicate = await
db.InstallationStateReports.SingleOrDefaultAsync(x => x.InstallationId == installationId &&
x.ReportedAt == request.ReportedAt, ct);
        if (duplicate is not null)
        {
            if (duplicate.AgentId != actor.AgentId ||
duplicate.InstalledProductVersionId != request.InstalledProductVersionId ||
duplicate.EffectiveConfigurationRevisionId != request.EffectiveConfigurationRevisionId ||
duplicate.ConfigurationHash != hash || duplicate.Status != request.Status)
            { throw new ApiConflictException("state-report-conflict", "A different
report already exists at this timestamp."); }
            await transaction.CommitAsync(ct); return Response(duplicate);
        }
    }

```

```

        }
        InstallationStateReport? latest = await
db.InstallationStateReports.SingleOrDefaultAsync(x => x.Id == installation.ReportedStateReportId,
ct);
        if (latest is not null && request.ReportedAt <= latest.ReportedAt) { throw new
ApiConflictException("state-report-stale", "The report is older than the accepted state. Refresh
and send a newer observation."); }
        InstallationStateReport report = new InstallationStateReport
        {
            InstallationId = installationId,
            AgentId = actor.AgentId,
            InstalledProductVersionId = request.InstalledProductVersionId,
            EffectiveConfigurationRevisionId =
request.EffectiveConfigurationRevisionId,
            ConfigurationHash = hash,
            Status = request.Status,
            ReportedAt = request.ReportedAt,
            ReceivedAt = clock.GetUtcNow()
        };
        db.InstallationStateReports.Add(report); await db.SaveChangesAsync(ct);
        installation.ReportedStateReportId = report.Id; installation.LastSeenAt = report.ReceivedAt;
        await InstallationStatePolicy.EnqueueAsync(db, installationId, report.ReceivedAt,
ct);
        audit.Record(db, actor.AgentId, "InstallationStateReported", installationId,
entityType: "Installation", actorType: "Agent", resourceId: report.Id);
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return
Response(report);
    }

    /// <summary>
    /// Opens a target artifact after checking agent ownership and current entitlement.
    /// </summary>
    public async Task<AgentArtifactContent> ArtifactAsync(AgentActor actor, Guid artifactId,
Cancellation.Token ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);
        var candidates = (from installation in db.Installations
                        join desired in db.DesiredStateRevisions on
installation.DesiredStateRevisionId equals desired.Id
                        join version in db.ProductVersions on
desired.ProductVersionId equals version.Id
                        join config in db.EffectiveConfigurationRevisions
on desired.EffectiveConfigurationRevisionId equals config.Id
                        where installation.AgentId == agent.Id &&
installation.CustomerId == agent.CustomerId && installation.Status == "Active"
                        && (version.PublishedInstallerArtifactId ==
artifactId || config.FileArtifactId == artifactId)
                        select new { Installation = installation, Desired
= desired });
        Guid? cursor = null;
        while (true)
        {
            var page = await candidates.Where(x => cursor == null ||
x.Installation.Id.CompareTo(cursor.Value) > 0).OrderBy(x =>
x.Installation.Id).Take(100).ToListAsync(ct);
            foreach (var item in page)

```



```

        {
            try { await policy.RequireEligibleAsync(db, item.Installation.Id,
ct); }

            catch (ApiConflictException) { continue; }

            StateTarget target = await policy.TargetAsync(tenant,
item.Installation, item.Desired.ProductVersionId, item.Desired.EffectiveConfigurationRevisionId,
ct);

            FileArtifact artifact = target.Installer.Id == artifactId ?
target.Installer : target.Configuration.FileArtifact;

            return new(await policy.OpenAsync(tenant.ArchivePath, artifact,
ct), artifact, target.Version.Id, target.Configuration.Id);
        }

        if (page.Count < 100) { break; }

        cursor = page[^1].Installation.Id; db.ChangeTracker.Clear();
    }

    throw new ApiAccessException(404);
}

/// <summary>
/// Loads an installation owned by the authenticated agent.
/// </summary>
private async Task<Installation> AgentInstallationAsync(TenantContext tenant, AgentActor
actor, Guid installationId, bool active, CancellationToken ct)
{
    Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);

    return await tenant.Database.Installations.SingleOrDefaultAsync(x => x.Id ==
installationId && x.AgentId == agent.Id && x.CustomerId == agent.CustomerId && (!active ||
x.Status == "Active"), ct) ?? throw new ApiAccessException(404);
}

/// <summary>
/// Supersedes obsolete deployment work after the desired target changes.
/// </summary>
internal static async Task SupersedeAsync(VendorDbContext db, Guid installation, Guid
desired, DateTimeOffset now, CancellationToken ct)
{
    foreach (Deployment? deployment in await db.Deployments.Where(x =>
x.InstallationId == installation && x.IsActive && x.DesiredStateRevisionId !=
desired).ToListAsync(ct))
    { deployment.IsActive = false; deployment.Status = "Superseded";
deployment.CompletedAt = now; }
}

/// <summary>
/// Maps stored entity data to the public response.
/// </summary>
private static DesiredStateResponse Response(DesiredStateRevision x) =>
new(x.InstallationId, x.ProductVersionId, x.EffectiveConfigurationRevisionId, x.UpdatedAt);

/// <summary>
/// Maps stored entity data to the public response.
/// </summary>
private static ReportedStateResponse Response(InstallationStateReport x) =>
new(x.InstallationId, x.InstalledProductVersionId, x.EffectiveConfigurationRevisionId,
x.ConfigurationHash, x.Status, x.ReportedAt);
}

```

IOperationalHistoryRepository.cs

```

using BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.OperationalHistoryContracts;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Vendor;
namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for operational history.
/// </summary>
public interface IOperationalHistoryRepository
{
    /// <summary>
    /// Validates, deduplicates, and stores an agent log batch within ingestion limits.
    /// </summary>
    Task SubmitAsync(AgentActor actor, SubmitAgentLogsRequest request, CancellationToken ct);

    /// <summary>
    /// Searches operational logs within the actor's permitted resource scope.
    /// </summary>
    Task<PagedResponse<LogResponse>> LogsAsync(TenantActor actor, PaginationQuery page,
LogSearchQuery query, Guid? installationRoute, Guid? agentRoute, CancellationToken ct);

    /// <summary>
    /// Builds an installation health overview from desired, reported, and deployment
evidence.
    /// </summary>
    Task<MonitoringOverviewResponse> OverviewAsync(TenantActor actor, PaginationQuery page,
CancellationToken ct);
}

```

IUserAdministrationRepository.cs

```

using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.UserAdministrationRequests;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for user administration.
/// </summary>
public interface IUserAdministrationRepository
{
    /// <summary>
    /// Returns a filtered page of user administration records.
    /// </summary>
    Task<PagedResponse<UserResponse>> PageAsync(TenantActor actor, PaginationQuery page,
UserQuery filter, CancellationToken ct);

    /// <summary>
    /// Returns the requested user administration record when available.
    /// </summary>
    Task<UserResponse?> GetAsync(TenantActor actor, Guid id, CancellationToken ct);

    /// <summary>
    /// Creates a user invitation within the caller's authorized scope.
    /// </summary>
}

```

```

        Task<UserResponse> InviteAsync(TenantActor actor, CreateUserRequest request,
CancellationToken ct);

        /// <summary>
        /// Updates the supplied fields of the selected user administration record.
        /// </summary>
        Task<UserResponse?> UpdateAsync(TenantActor actor, Guid id, UpdateUserRequest request,
CancellationToken ct);
    }

```

IVersionPublishingRepository.cs

```

using BridgeControl.DTOS.Vendor;
using BridgeControl.Services.Vendor;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Defines data access operations for version publishing.
/// </summary>
public interface IVersionPublishingRepository
{
    /// <summary>
    /// Validates and publishes a draft version with immutable release evidence.
    /// </summary>
    Task<ProductVersionDto> PublishAsync(TenantActor actor, Guid productId, Guid versionId,
CancellationToken ct);
}

```

LicenseRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.LicenseContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for license.
/// </summary>
public sealed class LicenseRepository(TenantOperationFactory operations, LicenseEligibility
eligibility, VendorSecurityAudit audit, TimeProvider clock) : ILicenseRepository
{
    private static readonly Expression<Func<License, LicenseResponse>> Response = x =>
new(x.Id, x.CustomerId, x.ProductId, x.LicenseReference, x.Status, x.ValidFrom, x.ValidUntil,
x.MaxInstallations);

    /// <summary>
    /// Builds the license query within the caller's permitted scope.
    /// </summary>

```

```

private static IQueryable<License> Scoped(TenantOperation operation, string permission)
{
    VendorDbContext db = operation.Database;
    IQueryable<Customer> customers = operation.Access.Customers(db, permission);
    IQueryable<Installation> installations = operation.Access.Installations(db,
permission);
    bool installationScoped = operation.Access.User.CustomerId is not null &&
!operation.Access.Roles.Contains("CustomerAdministrator");
    return db.Licenses.Where(x => customers.Any(customer => customer.Id ==
x.CustomerId) && (!installationScoped
|| db.InstallationLicenseAssignments.Any(a => a.LicenseId == x.Id &&
a.RemovedAt == null && installations.Any(i => i.Id == a.InstallationId))));
}

/// <summary>
/// Returns a filtered page of license records.
/// </summary>
public async Task<PagedResponse<LicenseResponse>> PageAsync(TenantActor actor,
PaginationQuery page, InstallationQuery filter, CancellationToken ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.read", false, ct);
    IQueryable<License> query = Scoped(operation, "licenses.read");
    if (filter.CustomerId is Guid customer) { query = query.Where(x => x.CustomerId ==
customer); }
    if (filter.ProductId is Guid product) { query = query.Where(x => x.ProductId ==
product); }
    if (!string.IsNullOrEmpty(filter.Status)) { string status =
filter.Status.Trim(); query = query.Where(x => x.Status == status); }
    return await new Repository<License, LicenseResponse>(query,
Response).PageAsync(page, x => x.Id, ct);
}

/// <summary>
/// Returns the requested license record when available.
/// </summary>
public async Task<LicenseResponse?> GetAsync(TenantActor actor, Guid id, Cancellation token
ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.read", false, ct);
    return await new Repository<License, LicenseResponse>(Scoped(operation,
"licenses.read").Where(x => x.Id == id), Response).SingleOrDefaultAsync(ct);
}

/// <summary>
/// Creates a license record from the validated request.
/// </summary>
public async Task<LicenseResponse> CreateAsync(TenantActor actor, CreateLicenseRequest
request, CancellationToken ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.manage", true, ct);
    VendorDbContext db = operation.Database;
    if (!await operation.Access.Customers(db, "licenses.manage").AnyAsync(x => x.Id ==
request.CustomerId && x.Status == "Active", ct)
|| !await db.Products.AnyAsync(x => x.Id == request.ProductId && x.Status
== "Active", ct)) { throw new ApiAccessException(404); }
    License license = new()
    {
        CustomerId = request.CustomerId,
        ProductId = request.ProductId,

```

```

        LicenseReference = request.LicenseReference,
        ValidFrom = request.ValidFrom,
        ValidUntil = request.ValidUntil,
        MaxInstallations = request.MaxInstallations
    };
    db.Licenses.Add(license); audit.Record(db, actor.UserId, "LicenseCreated",
license.Id, entityType: "License"); await db.SaveChangesAsync(ct);
    return await db.Licenses.Where(x => x.Id ==
license.Id).Select(Response).SingleAsync(ct);
}

/// <summary>
/// Updates the supplied fields of the selected license record.
/// </summary>
public async Task<LicenseResponse?> UpdateAsync(TenantActor actor, Guid id,
UpdateLicenseRequest request, CancellationToken ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.manage", true, ct);
    VendorDbContext db = operation.Database;
    License? license = await Scoped(operation,
"licenses.manage").SingleOrDefaultAsync(x => x.Id == id, ct);
    if (license is null) { return null; }
    if (request.ReferenceProvided) { license.LicenseReference =
request.LicenseReference!; }
    if (request.FromProvided) { license.ValidFrom = request.ValidFrom!.Value; }
    if (request.UntilProvided) { license.ValidUntil = request.ValidUntil; }
    if (request.MaximumProvided) { license.MaxInstallations =
request.MaxInstallations; }
    if (license.ValidUntil <= license.ValidFrom) { throw new
LicenseValidationException("ValidUntil must be later than ValidFrom."); }
    await CheckCapacityAsync(db, license, ct);
    db.ChangeTracker.DetectChanges();
    if (db.Entry(license).State == EntityState.Modified)
    {
        audit.Record(db, actor.UserId, "LicenseUpdated", id, entityType:
"License"); await db.SaveChangesAsync(ct);
    }
    return await db.Licenses.Where(x => x.Id == id).Select(Response).SingleAsync(ct);
}

/// <summary>
/// Changes the active status of the selected license record.
/// </summary>
public async Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active,
CancellationToken ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.manage", true, ct);
    VendorDbContext db = operation.Database;
    License? license = await Scoped(operation, "licenses.manage").Include(x =>
x.Customer).Include(x => x.Product).SingleOrDefaultAsync(x => x.Id == id, ct);
    if (license is null) { return false; }
    string status = active ? "Active" : "Disabled"; if (license.Status == status) {
return true; }
    if (active)
    {
        if (license.Customer.Status != "Active" || license.Product.Status !=
"Active") { throw new ApiConflictException("license-unavailable", "The license's customer or
product is disabled."); }
        await CheckCapacityAsync(db, license, ct);
    }
}

```

```

        license.Status = status;
        audit.Record(db, actor.UserId, active ? "LicenseReactivated" :
"LicenseDeactivated", id, entityType: "License"); await db.SaveChangesAsync(ct); return true;
    }

    /// <summary>
    /// Assigns or removes an installation license while enforcing ownership and capacity.
    /// </summary>
    public async Task<bool> AssignAsync(TenantActor actor, Guid installationId, Guid?
licenseId, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"licenses.manage", true, ct);
        VendorDbContext db = operation.Database;
        Installation? installation = await operation.Access.Installations(db,
"licenses.manage").Include(x => x.Customer).Include(x => x.Product).SingleOrDefaultAsync(x =>
x.Id == installationId, ct);
        if (installation is null) { return false; }
        InstallationLicenseAssignment? current = await
db.InstallationLicenseAssignments.SingleOrDefaultAsync(x => x.InstallationId == installationId &&
x.RemovedAt == null, ct);
        if (licenseId is Guid id)
        {
            License license = await Scoped(operation,
"licenses.manage").SingleOrDefaultAsync(x => x.Id == id, ct) ?? throw new
ApiAccessException(404);
            if (installation.Status != "Active" || installation.Customer.Status !=
"Active" || installation.Product.Status != "Active")
            { throw new ApiConflictException("installation-unavailable", "An active
installation, customer and product are required."); }
            await eligibility.RequireAssignmentAsync(db, installation, license, ct);
        }
        if (current?.LicenseId == licenseId) { return true; }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        DateTimeOffset now = clock.UtcNow();
        if (current is not null)
        {
            current.RemovedAt = now; current.RemovedBy = actor.UserId;
            audit.Record(db, actor.UserId, "InstallationLicenseRemoved",
installationId, entityType: "Installation", resourceId: current.LicenseId);
            // Release the filtered unique index before inserting a replacement, within
the same transaction.
            await db.SaveChangesAsync(ct);
        }
        if (licenseId is Guid assigned)
        {
            db.InstallationLicenseAssignments.Add(new()
            {
                InstallationId = installationId,
                LicenseId = assigned,
                CustomerId = installation.CustomerId,
                ProductId = installation.ProductId,
                AssignedAt = now,
                AssignedBy = actor.UserId
            });
            audit.Record(db, actor.UserId, "InstallationLicenseAssigned",
installationId, entityType: "Installation", resourceId: assigned);
        }
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return true;
    }
}

```

```

    /// <summary>
    /// Rejects license changes that would violate active installation capacity.
    /// </summary>
    private async Task CheckCapacityAsync(VendorDbContext db, License license,
    CancellationToken ct)
    {
        if (license.MaxInstallations is int maximum && await
    eligibility.ActiveCountAsync(db, license.Id, null, ct) > maximum)
        { throw new ApiConflictException("license-capacity", "Capacity cannot be below the
    current active installation count."); }
    }
}

```

OperationalHistoryRepository.cs

```

using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.AgentDTOS.OperationalHistoryContracts;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.OperationalHistoryContracts;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Repositories.Shared;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;
using Agent = BridgeControl.Models.AgentModels.Agent;

namespace BridgeControl.Repositories.Vendor;
/// <summary>
/// Implements data access operations for operational history.
/// </summary>
public sealed class OperationalHistoryRepository(TenantOperationFactory operations,
PlatformDbContext platform, ITenantResolver resolver,
    AgentIdentityService identity, IOptions<OperationalHistoryOptions> options, TimeProvider
    clock) : IOperationalHistoryRepository
{
    /// <summary>
    /// Validates, deduplicates, and stores an agent log batch within ingestion limits.
    /// </summary>
    public async Task SubmitAsync(AgentActor actor, SubmitAgentLogsRequest request,
    CancellationToken ct)
    {
        DateTimeOffset now = clock.GetUtcNow();
        if (request.Entries is null || request.Entries.Count is < 1 or > 100) { throw new
    AgentValidationException("Submit between 1 and 100 log entries."); }
        foreach (AgentLogEntryRequest? e in request.Entries)
        {
            if (e is null || !LogSeverities.Values.Contains(e.Severity) ||
    string.IsNullOrEmpty(e.Source) || e.Source.Length > 100
                || string.IsNullOrEmpty(e.Message) || e.Message.Length > 2000
                || e.Source.Any(char.IsControl)

```

```

        || e.CreatedAt < now.AddDays(-options.Value.LogDays) || e.CreatedAt
> now.AddMinutes(5))
        { throw new AgentValidationException("Log severity, source, message or
timestamp is invalid."); }
    }
    await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
    await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct) ?? throw new ApiAccessException(401);
    Agent agent = await identity.CurrentAsync(tenant, actor, ct) ?? throw new
ApiAccessException(401);
    VendorDbContext db = tenant.Database;
    await using IDbContextTransaction tx = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
    Guid[] installations = request.Entries.Where(x => x.InstallationId !=
null).Select(x => x.InstallationId!.Value).Distinct().ToArray();
    if (await db.Installations.CountAsync(x => installations.Contains(x.Id) &&
x.AgentId == agent.Id && x.CustomerId == agent.CustomerId, ct) != installations.Length)
    { throw new ApiAccessException(404); }
    LogIngestionWindow? window = await db.LogIngestionWindows.SingleOrDefaultAsync(x
=> x.AgentId == agent.Id, ct);
    if (window is null) { window = new() { AgentId = agent.Id, StartedAt = now };
db.LogIngestionWindows.Add(window); }
    if (window.StartedAt.AddMinutes(1) <= now) { window.StartedAt = now;
window.Batches = 0; window.Entries = 0; }
    if (window.Batches >= 10 || window.Entries + request.Entries.Count > 500) { throw
new AgentRateLimitException(); }
    window.Batches++; window.Entries += request.Entries.Count;
    OperationalLog[] proposed = request.Entries.Select(e =>
    {
        bool sensitive = OperationalRedaction.IsSensitive(e.Source) ||
OperationalRedaction.IsSensitive(e.Message);
        string canonical = JsonSerializer.Serialize(new { e.InstallationId,
e.Severity, e.Source, e.Message, CreatedAt = e.CreatedAt.ToUniversalTime() });
        return new OperationalLog
        {
            AgentId = agent.Id,
            InstallationId = e.InstallationId,
            Severity = e.Severity,
            Source = sensitive ? "Redacted" : e.Source,
            Message = sensitive ? "[Sensitive log content removed]" :
e.Message,
            IsSensitive = sensitive,
            Fingerprint =
Convert.ToHexString(SHA256.HashData(Encoding.UTF8.GetBytes(canonical))),
            CreatedAt = e.CreatedAt,
            ReceivedAt = now
        };
    }).DistinctBy(x => x.Fingerprint).ToArray();
    string[] hashes = proposed.Select(x => x.Fingerprint).ToArray();
    string[] existing = await db.OperationalLogs.Where(x => x.AgentId == agent.Id &&
hashes.Contains(x.Fingerprint)).Select(x => x.Fingerprint).ToArrayAsync(ct);
    OperationalLog[] additions = proposed.Where(x =>
!existing.Contains(x.Fingerprint)).ToArray();
    if (additions.Length > 0 && await db.OperationalLogs.CountAsync(ct) +
additions.Length > options.Value.MaxLogsPerVendor)
    { throw new ApiConflictException("log-storage-limit", "Operational log storage is
full; retry after retention cleanup."); }
    db.OperationalLogs.AddRange(additions); await db.SaveChangesAsync(ct); await
tx.CommitAsync(ct);
}

```



```

    /// <summary>
    /// Searches operational logs within the actor's permitted resource scope.
    /// </summary>
    public async Task<PagedResponse<LogResponse>> LogsAsync(TenantActor actor, PaginationQuery
page, LogSearchQuery search, Guid? installationRoute, Guid? agentRoute, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor, "logs.read",
false, ct); VendorDbContext db = op.Database;
        IQueryable<Installation> installations = op.Access.Installations(db, "logs.read");
        IQueryable<Customer> customers = op.Access.Customers(db, "logs.read");
        bool wholeCustomer = op.Access.User.CustomerId == null ||
op.Access.Roles.Contains("CustomerAdministrator");
        bool sensitive = op.Access.User.CustomerId == null &&
op.Access.Roles.Contains("VendorAdministrator");
        IQueryable<Agent> agents = db.Agents.Where(a => customers.Any(c => c.Id ==
a.CustomerId) && (wholeCustomer || installations.Any(i => i.AgentId == a.Id)));
        if (installationRoute.HasValue && !await installations.AnyAsync(x => x.Id ==
installationRoute, ct)) { throw new ApiAccessException(404); }
        if (agentRoute.HasValue && !await agents.AnyAsync(x => x.Id == agentRoute, ct)) {
throw new ApiAccessException(404); }
        IQueryable<OperationalLog> query = db.OperationalLogs.Where(l => (sensitive ||
!l.IsSensitive) && agents.Any(a => a.Id == l.AgentId)
        && (l.InstallationId == null ? wholeCustomer : installations.Any(i => i.Id
== l.InstallationId)));
        if (installationRoute.HasValue) { query = query.Where(x => x.InstallationId ==
installationRoute); }
        if (agentRoute.HasValue) { query = query.Where(x => x.AgentId == agentRoute); }
        if (search.InstallationId.HasValue) { query = query.Where(x => x.InstallationId ==
search.InstallationId); }
        if (search.AgentId.HasValue) { query = query.Where(x => x.AgentId ==
search.AgentId); }
        if (search.Severity != null) { query = query.Where(x => x.Severity ==
search.Severity); }
        if (search.From.HasValue) { query = query.Where(x => x.CreatedAt >= search.From);
}

        if (search.To.HasValue) { query = query.Where(x => x.CreatedAt < search.To); }
        return await new Repository<OperationalLog, LogResponse>(query, x => new(x.Id,
x.AgentId, x.InstallationId, x.Severity, x.Source, x.Message, x.CreatedAt)).PageAsync(page, x =>
x.Id, ct);
    }

    /// <summary>
    /// Builds an installation health overview from desired, reported, and deployment
evidence.
    /// </summary>
    public async Task<MonitoringOverviewResponse> OverviewAsync(TenantActor actor,
PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"monitoring.read", true, ct); VendorDbContext db = op.Database;
        await using IDbContextTransaction tx = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        DateTimeOffset now = clock.GetUtcNow(), online = now.AddMinutes(-5), fresh =
now.AddMinutes(-15);
        IQueryable<Installation> scoped = op.Access.Installations(db, "monitoring.read");
        IQueryable<HealthRow> health = from i in scoped
                                let agent =
db.Agents.FirstOrDefault(a => a.Id == i.AgentId)
                                let desired =
db.DesiredStateRevisions.FirstOrDefault(d => d.Id == i.DesiredStateRevisionId)

```

```

let report =
db.InstallationStateReports.FirstOrDefault(r => r.Id == i.ReportedStateReportId && r.AgentId ==
i.AgentId)

let license =
db.InstallationLicenseAssignments.Where(a => a.InstallationId == i.Id && a.RemovedAt ==
null).Select(a => a.License).FirstOrDefault()

let deployment =
db.Deployments.Where(d => d.InstallationId == i.Id).OrderByDescending(d =>
d.RequestedAt).ThenByDescending(d => d.Id).FirstOrDefault()

let current = report !=
null && report.ReportedAt >= fresh && report.Status != "Unknown"

let matching = current &&
desired != null && report!.Status == "Running" && report.InstalledProductVersionId ==
desired.ProductVersionId

&&
report.EffectiveConfigurationRevisionId == desired.EffectiveConfigurationRevisionId
&&
db.EffectiveConfigurationRevisions.Any(c => c.Id == desired.EffectiveConfigurationRevisionId &&
c.FileArtifact.Sha256 == report.ConfigurationHash)

select new HealthRow
{
    Id = i.Id,
    Connectivity =
agent == null || agent.LastSeenAt == null ? "Unknown" : agent.Status != "Active" ||
agent.LastSeenAt < fresh ? "Offline" : agent.LastSeenAt < online ? "Stale" : "Online",
    Freshness = report
== null ? "Unknown" : report.ReportedAt < fresh ? "Stale" : report.Status == "Unknown" ?
"Unknown" : "Fresh",
    Drift = desired ==
null || !current ? "Unknown" : matching ? "Matching" : "Drifted",
    Configuration =
db.EffectiveConfigurationWork.Any(w => w.InstallationId == i.Id && (desired == null ||
w.ProductVersionId == desired.ProductVersionId) && w.Status == "Failed") ? "Failed"
:
db.EffectiveConfigurationWork.Any(w => w.InstallationId == i.Id && (desired == null ||
w.ProductVersionId == desired.ProductVersionId) && w.Status == "Pending") ? "Pending" :
i.CurrentEffectiveConfigurationRevisionId == null ? "Unknown" : "Ready",
    License = i.Status
!= "Active" || i.Customer.Status != "Active" || i.Product.Status != "Active" ?
"InstallationUnavailable"
: agent ==
null || agent.Status != "Active" ? "AgentUnavailable" : license == null ? "LicenseMissing"
:
license.CustomerId != i.CustomerId || license.ProductId != i.ProductId || license.Status !=
"Active" || license.ValidFrom > now || license.ValidUntil <= now ? "LicenseUnavailable"
:
license.MaxInstallations != null && db.InstallationLicenseAssignments.Count(a => a.LicenseId ==
license.Id && a.RemovedAt == null && a.Installation.Status == "Active") >
license.MaxInstallations ? "LicenseCapacity" : "Eligible",
    Deployment =
deployment == null ? "None" : deployment.Status
};

int total = await health.CountAsync(ct);
int on = await health.CountAsync(x => x.Connectivity == "Online", ct), stale =
await health.CountAsync(x => x.Connectivity == "Stale", ct);
int off = await health.CountAsync(x => x.Connectivity == "Offline", ct), unknown =
await health.CountAsync(x => x.Connectivity == "Unknown", ct);
int drift = await health.CountAsync(x => x.Drift == "Drifted", ct), ineligible =
await health.CountAsync(x => x.License != "Eligible", ct);
int failed = await health.CountAsync(x => x.Deployment == "Failed", ct), pending =
await health.CountAsync(x => x.Deployment == "Pending", ct);
int configFailed = await health.CountAsync(x => x.Configuration == "Failed", ct);

```

```

        IQueryable<HealthRow> attention = health.Where(x => x.Connectivity != "Online" ||
x.Freshness != "Fresh" || x.Drift != "Matching" || x.Configuration != "Ready" || x.License !=
"Eligible" || (x.Deployment != "None" && x.Deployment != "Reconciled"));
        int attentionCount = await attention.CountAsync(ct);
        PagedResponse<InstallationHealthResponse> items = await new Repository<HealthRow,
InstallationHealthResponse>(health, x => new(x.Id, x.Connectivity, x.Freshness, x.Drift,
x.Configuration, x.License, x.Deployment,
        x.Connectivity != "Online" || x.Freshness != "Fresh" || x.Drift !=
"Matching" || x.Configuration != "Ready" || x.License != "Eligible" || (x.Deployment != "None" &&
x.Deployment != "Reconciled"))).PageAsync(page, x => x.Id, ct);
        await tx.CommitAsync(ct); return new(now, total, on, stale, off, unknown, drift,
ineligible, failed, pending, configFailed, attentionCount, items);
    }

    /// <summary>
    /// Collects installation evidence used to calculate the monitoring health response.
    /// </summary>
    /// <remarks>
    ///     I don't generally like nested classes and this one could technically be moved out,
    ///     but it's a private implementation detail of the repository and I want to keep it
scoped to this file,
    ///     to avoid usding it elsewhere and having to maintain it as a public contract.
    ///     It's also a simple data holder with no behavior, so it doesn't need to be a full
record type.
    /// </remarks>
    private sealed class HealthRow
    {
        public Guid Id { get; set; }
        public string Connectivity { get; set; } = "";
        public string Freshness { get; set; } = "";
        public string Drift { get; set; } = "";
        public string Configuration { get; set; } = "";
        public string License { get; set; } = "";
        public string Deployment { get; set; } = "";
    }
}

```

ProductArtifactRepository.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for product artifact.
/// </summary>
public class ProductArtifactRepository(TenantOperationFactory operations, VendorArtifactArchive
archive, VendorSecurityAudit audit, TimeProvider clock)
{
    /// <summary>
    /// Returns the requested product artifact record when available.
    /// </summary>
    public async Task<FileArtifactDto?> GetAsync(TenantActor actor, Guid productId, Guid
versionId, string kind, CancellationToken ct)

```

```

        {
            await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
            await VersionAsync(op, productId, versionId, false, ct);
            FileArtifact? artifact = await ArtifactAsync(op, versionId, kind, ct);
            return artifact is null ? null : Response(artifact);
        }

    /// <summary>
    /// Opens verified content for a version artifact within the actor's authorized scope.
    /// </summary>
    public async Task<(Stream Stream, FileArtifactDto Metadata)> ContentAsync(TenantActor
actor, Guid productId, Guid versionId, string kind, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        await VersionAsync(op, productId, versionId, false, ct);
        FileArtifact artifact = await ArtifactAsync(op, versionId, kind, ct) ?? throw new
ApiAccessException(404);
        return (await archive.OpenAsync(op.Tenant.ArchivePath, artifact, ct),
Response(artifact));
    }

    /// <summary>
    /// Validates and attaches an artifact to a draft product version.
    /// </summary>
    public async Task<FileArtifactDto> PutAsync(TenantActor actor, Guid productId, Guid
versionId, string kind, string fileName,
        Stream source, Func<CancellationToken, Task> validateEnd, CancellationToken ct)
    {
        string archivePath;
        // Authorize before reading bytes; do not hold the administrative write lock
during network uploads.
        await using (TenantOperation read = await operations.OpenAsync(actor,
"products.manage", false, ct))
        {
            await VersionAsync(read, productId, versionId, true, ct);
            archivePath = read.Tenant.ArchivePath;
        }
        using CancellationTokenSource timeout =
CancellationTokenSource.CreateLinkedTokenSource(ct);
        timeout.CancelAfter(archive.UploadTimeout);
        await using StagedArtifact staged = await archive.StageAsync(archivePath, kind,
fileName, source, timeout.Token);
        await validateEnd(timeout.Token);
        // Recheck current identity, permission, parent, lifecycle and draft state after
the upload.
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, timeout.Token);
        ProductVersion version = await VersionAsync(op, productId, versionId, true,
timeout.Token);
        if (!string.Equals(archivePath, op.Tenant.ArchivePath, StringComparison.Ordinal))
        { throw new ApiConflictException("archive-changed", "Retry the upload."); }
        FileArtifact? existing = await ArtifactAsync(op, versionId, kind, timeout.Token);
        if (existing is not null && existing.Sha256 == staged.Sha256 && existing.SizeBytes
== staged.SizeBytes
            && existing.FileName == staged.FileName && existing.ContentType ==
staged.ContentType) { return Response(existing); }
        VendorDbContext db = op.Database;
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(timeout.Token);
        if (existing is not null)

```

```

        {
            ProductVersionArtifact link = await
db.ProductVersionArtifacts.SingleAsync(x => x.ProductVersionId == versionId && x.ArtifactType ==
kind, timeout.Token);
            db.ProductVersionArtifacts.Remove(link);
            await db.SaveChangesAsync(timeout.Token);
        }
        FileArtifact artifact = new()
        {
            InstallerIdentityJson = staged.InstallerIdentityJson,
            Id = Guid.NewGuid(),
            ArtifactType = kind,
            FileName = staged.FileName,
            ContentType = staged.ContentType,
            StorageIdentifier = staged.StorageIdentifier,
            Sha256 = staged.Sha256,
            SizeBytes = staged.SizeBytes,
            CreatedAt = clock.GetUtcNow()
        };
        db.FileArtifacts.Add(artifact);
        db.ProductVersionArtifacts.Add(new() { ProductVersionId = versionId,
FileArtifactId = artifact.Id, ArtifactType = kind });
        db.Entry(version).Property(x => x.Version).IsModified = true;
        if (kind == "Schema") { db.EffectiveConfigurationSignals.Add(new() {
ProductVersionId = versionId, CreatedAt = clock.GetUtcNow() }); }
        audit.Record(db, actor.UserId, kind + "Uploaded", versionId, entityType:
"ProductVersion", resourceId: artifact.Id);
        staged.Promote();
        await db.SaveChangesAsync(timeout.Token);
        await transaction.CommitAsync(timeout.Token);
        return Response(artifact);
    }

    /// <summary>
    /// Removes an artifact attachment from a draft product version.
    /// </summary>
    public async Task DeleteAsync(TenantActor actor, Guid productId, Guid versionId, string
kind, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        ProductVersion version = await VersionAsync(op, productId, versionId, true, ct);
        ProductVersionArtifact? link = await
op.Database.ProductVersionArtifacts.SingleOrDefaultAsync(x => x.ProductVersionId == versionId &&
x.ArtifactType == kind, ct);
        if (link is null) { return; }
        // Unlink the draft slot, retaining immutable metadata/blobs and any historical
references.
        op.Database.ProductVersionArtifacts.Remove(link);
        op.Database.Entry(version).Property(x => x.Version).IsModified = true;
        if (kind == "Schema") { op.Database.EffectiveConfigurationSignals.Add(new() {
ProductVersionId = versionId, CreatedAt = clock.GetUtcNow() }); }
        audit.Record(op.Database, actor.UserId, kind + "Removed", versionId, entityType:
"ProductVersion", resourceId: link.FileArtifactId);
        await op.Database.SaveChangesAsync(ct);
    }

    /// <summary>
    /// Loads an authorized product version and optionally requires it to remain a draft.
    /// </summary>
    private static async Task<ProductVersion> VersionAsync(TenantOperation op, Guid productId,
Guid versionId, bool draft, CancellationToken ct)

```

```

        {
            ProductVersion version = await op.Database.ProductVersions.SingleOrDefaultAsync(x
=> x.ProductId == productId && x.Id == versionId, ct) ?? throw new ApiAccessException(404);
            if (draft) { ProductRepository.RequireDraft(version); }
            return version;
        }

        /// <summary>
        /// Loads the selected artifact attachment and its file metadata.
        /// </summary>
        private static Task<FileArtifact?> ArtifactAsync(TenantOperation op, Guid versionId,
string kind, CancellationToken ct)
            => op.Database.ProductVersionArtifacts.Where(x => x.ProductVersionId == versionId
&& x.ArtifactType == kind).Select(x => x.FileArtifact).SingleOrDefaultAsync(ct);

        /// <summary>
        /// Maps stored entity data to the public response.
        /// </summary>
        private static FileArtifactDto Response(FileArtifact x) => new()
        { Id = x.Id, ArtifactType = x.ArtifactType, FileName = x.FileName, ContentType =
x.ContentType, Sha256 = x.Sha256, SizeBytes = x.SizeBytes, CreatedAt = x.CreatedAt };
    }

```

ProductRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOs.Shared;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for product.
/// </summary>
public class ProductRepository(TenantOperationFactory operations, VendorSecurityAudit audit,
TimeProvider clock)
{
    private static readonly Expression<Func<Product, ProductDto>> ProductResponse = x => new()
    { Id = x.Id, Name = x.Name, Description = x.Description, Status = x.Status, CreatedAt =
x.CreatedAt };
    private static readonly Expression<Func<ProductVersion, ProductVersionDto>>
VersionResponse = x => new()
    { Id = x.Id, ProductId = x.ProductId, Version = x.Version, Status = x.Status, CreatedAt =
x.CreatedAt, PublishedAt = x.PublishedAt };

    /// <summary>
    /// Returns a filtered page of product records.
    /// </summary>
    public async Task<PagedResponse<ProductDto>> PageAsync(TenantActor actor, PaginationQuery
page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        return await new Repository<Product, ProductDto>(op.Database.Products,
ProductResponse).PageAsync(page, x => x.Id, ct);
    }

```

```

    }

    /// <summary>
    /// Returns the requested product record when available.
    /// </summary>
    public async Task<ProductDto?> GetAsync(TenantActor actor, Guid id, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        return await op.Database.Products.Where(x => x.Id ==
id).Select(ProductResponse).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Creates a product record from the validated request.
    /// </summary>
    public async Task<ProductDto> CreateAsync(TenantActor actor, CreateProductRequest request,
CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        Product product = new() { Id = Guid.NewGuid(), Name = request.Name.Trim(),
Description = request.Description?.Trim(), Status = "Active", CreatedAt = clock.GetUtcNow() };
        op.Database.Products.Add(product);
        audit.Record(op.Database, actor.UserId, "ProductCreated", product.Id, entityType:
"Product");
        await op.Database.SaveChangesAsync(ct);
        return await op.Database.Products.Where(x => x.Id ==
product.Id).Select(ProductResponse).SingleAsync(ct);
    }

    /// <summary>
    /// Updates the supplied fields of the selected product record.
    /// </summary>
    public async Task<ProductDto?> UpdateAsync(TenantActor actor, Guid id,
UpdateProductRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        Product? product = await op.Database.Products.SingleOrDefaultAsync(x => x.Id ==
id, ct);
        if (product is null) { return null; }
        if (request.NameProvided) { product.Name = request.Name!; }
        if (request.DescriptionProvided) { product.Description = request.Description; }
        if (op.Database.ChangeTracker.HasChanges())
        {
            audit.Record(op.Database, actor.UserId, "ProductUpdated", id, entityType:
"Product");
            await op.Database.SaveChangesAsync(ct);
        }
        return await op.Database.Products.Where(x => x.Id ==
id).Select(ProductResponse).SingleAsync(ct);
    }

    /// <summary>
    /// Changes the active status of the selected product record.
    /// </summary>
    public async Task<bool> SetActiveAsync(TenantActor actor, Guid id, bool active,
CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);

```

```

        Product? product = await op.Database.Products.SingleOrDefaultAsync(x => x.Id ==
id, ct);

        if (product is null) { return false; }
        string status = active ? "Active" : "Disabled";
        if (product.Status == status) { return true; }
        product.Status = status;
        audit.Record(op.Database, actor.UserId, active ? "ProductReactivated" :
"ProductDeactivated", id, entityType: "Product");
        await op.Database.SaveChangesAsync(ct); return true;
    }

    /// <summary>
    /// Returns a page of versions belonging to the authorized product.
    /// </summary>
    public async Task<PagedResponse<ProductVersionDto>> VersionsAsync(TenantActor actor, Guid
productId, PaginationQuery page, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        if (!await op.Database.Products.AnyAsync(x => x.Id == productId, ct)) { throw new
ApiAccessException(404); }
        return await new Repository<ProductVersion,
ProductVersionDto>(op.Database.ProductVersions.Where(x => x.ProductId == productId),
VersionResponse).PageAsync(page, x => x.Id, ct);
    }

    /// <summary>
    /// Returns the selected version within its authorized parent product.
    /// </summary>
    public async Task<ProductVersionDto?> VersionAsync(TenantActor actor, Guid productId, Guid
versionId, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        return await op.Database.ProductVersions.Where(x => x.ProductId == productId &&
x.Id == versionId).Select(VersionResponse).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Creates a draft version for the authorized product.
    /// </summary>
    public async Task<ProductVersionDto> CreateVersionAsync(TenantActor actor, Guid productId,
CreateProductVersionRequest request, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        if (!await op.Database.Products.AnyAsync(x => x.Id == productId, ct)) { throw new
ApiAccessException(404); }
        ProductVersion version = new() { Id = Guid.NewGuid(), ProductId = productId,
Version = request.Version.Trim(), Status = "Draft", CreatedAt = clock.GetUtcNow() };
        op.Database.ProductVersions.Add(version);
        audit.Record(op.Database, actor.UserId, "ProductVersionCreated", version.Id,
entityType: "ProductVersion");
        await op.Database.SaveChangesAsync(ct);
        return await op.Database.ProductVersions.Where(x => x.Id ==
version.Id).Select(VersionResponse).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Updates the supplied fields of an authorized draft version.
    /// </summary>

```



```

        public async Task<ProductVersionDto?> UpdateVersionAsync(TenantActor actor, Guid
productId, Guid versionId, UpdateProductVersionRequest request, CancellationToken ct)
        {
            await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
            ProductVersion? version = await op.Database.ProductVersions.SingleOrDefaultAsync(x
=> x.Id == versionId && x.ProductId == productId, ct);
            if (version is null) { return null; }
            RequireDraft(version);
            if (request.VersionProvided && request.Version != version.Version)
            {
                version.Version = request.Version!;
                audit.Record(op.Database, actor.UserId, "ProductVersionUpdated", versionId,
entityType: "ProductVersion");
                await op.Database.SaveChangesAsync(ct);
            }
            return await op.Database.ProductVersions.Where(x => x.Id ==
versionId).Select(VersionResponse).SingleAsync(ct);
        }

        /// <summary>
        /// Rejects changes to a product version that is no longer a draft.
        /// </summary>
        internal static void RequireDraft(ProductVersion version)
        {
            if (version.Status != "Draft" || version.PublishedAt is not null)
            { throw new ApiConflictException("immutable-version", "Only draft versions can be
changed."); }
        }
    }
}

```

UserAdministrationRepository.cs

```

using System.Linq.Expressions;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor;
using BridgeControl.DTOS.Vendor.UserAdministrationRequests;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for user administration.
/// </summary>
public class UserAdministrationRepository(TenantOperationFactory operations, RoleGrantPolicy
roles,
    AccountInvitationService invitations, TenantPasswordService credentials,
VendorSecurityAudit audit, TimeProvider clock) : IUserAdministrationRepository
{
    public static readonly Expression<Func<UserAccount, UserResponse>> Response = user => new
UserResponse
    {
        Id = user.Id,

```

```

        CustomerId = user.CustomerId,
        Email = user.Email,
        Status = user.Status,
        CreatedAt = user.CreatedAt,
        Roles = user.UserRoles.OrderBy(x => x.Role.Name).Select(x =>
x.Role.Name).ToArray()
    };

    /// <summary>
    /// Returns a filtered page of user administration records.
    /// </summary>
    public async Task<PagedResponse<UserResponse>> PageAsync(TenantActor actor,
PaginationQuery page, UserQuery filter, CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.read", false, ct);
        IQueryable<UserAccount> query = operation.Access.Users(operation.Database);
        if (filter.CustomerId is not null) { query = query.Where(x => x.CustomerId ==
filter.CustomerId); }
        if (!string.IsNullOrEmpty(filter.Email)) { string email =
filter.Email.Trim(); query = query.Where(x => x.Email.Contains(email)); }
        if (!string.IsNullOrEmpty(filter.Status)) { query = query.Where(x => x.Status
== filter.Status); }
        return await new Repository<UserAccount, UserResponse>(query,
Response).PageAsync(page, x => x.Id, ct);
    }

    /// <summary>
    /// Returns the requested user administration record when available.
    /// </summary>
    public async Task<UserResponse?> GetAsync(TenantActor actor, Guid id, CancellationToken
ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.read", false, ct);
        return await new Repository<UserAccount,
UserResponse>(operation.Access.Users(operation.Database).Where(x => x.Id == id),
Response).SingleOrDefaultAsync(ct);
    }

    /// <summary>
    /// Creates a user invitation within the caller's authorized scope.
    /// </summary>
    public async Task<UserResponse> InviteAsync(TenantActor actor, CreateUserRequest request,
CancellationToken ct)
    {
        await using TenantOperation operation = await operations.OpenAsync(actor,
"users.manage", true, ct);
        VendorDbContext db = operation.Database;
        List<Role> requestedRoles = await roles.ValidateAsync(db, operation.Access,
request.CustomerId, request.RoleIds, ct);
        if (request.CustomerId is not null && !await
operation.Access.Customers(db).AnyAsync(x => x.Id == request.CustomerId && x.Status == "Active",
ct))
        {
            throw new ApiAccessException(404);
        }
        string email = AccountValidation.NormalizeEmail(request.Email);
        UserAccount? user = await db.UserAccounts.Include(x =>
x.UserRoles).SingleOrDefaultAsync(x => x.Email == email, ct);
        if (user is not null)
        {
            if (!await operation.Access.Users(db, "users.manage").AnyAsync(x => x.Id ==
user.Id, ct)) { throw new ApiAccessException(404); }

```

```

        if (user.Status != "Invited" || user.CustomerId != request.CustomerId ||
!user.UserRoles.Select(x => x.RoleId).ToHashSet().SetEquals(request.RoleIds))
        { throw new ApiConflictException("email-in-use", "An account with that
email already exists."); }
    }
    AccountActivation activation;
    await using (IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct))
    {
        if (user is null)
        {
            user = new UserAccount { Id = Guid.NewGuid(), CustomerId =
request.CustomerId, Email = email, Status = "Invited", CreatedAt = clock.GetUtcNow() };
            foreach (Role role in requestedRoles) { user.UserRoles.Add(new
UserRole { Id = Guid.NewGuid(), UserId = user.Id, RoleId = role.Id }); }
            db.UserAccounts.Add(user);
        }
        activation = await invitations.EnsureAsync(db, actor.VendorId, user,
actor.UserId, "UserInvited", "User", ct);
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
    }
    await DeliverAsync(operation, actor, user, activation, ct);
    return await db.UserAccounts.Where(x => x.Id ==
user.Id).Select(Response).SingleAsync(ct);
}

/// <summary>
/// Updates the supplied fields of the selected user administration record.
/// </summary>
public async Task<UserResponse?> UpdateAsync(TenantActor actor, Guid id, UpdateUserRequest
request, CancellationToken ct)
{
    await using TenantOperation operation = await operations.OpenAsync(actor,
"users.manage", true, ct);
    VendorDbContext db = operation.Database;
    UserAccount? user = await operation.Access.Users(db,
"users.manage").SingleOrDefaultAsync(x => x.Id == id, ct);
    if (user is null) { return null; }
    if (!request.EmailProvided || user.Email == request.Email)
    {
        // Retrying a completed email update can finish its interrupted delivery
        without issuing a new token.
        if (request.EmailProvided && user.Status == "Invited")
        {
            AccountActivation? pending = await
db.AccountActivations.SingleOrDefaultAsync(x => x.UserId == id && x.UsedAt == null && x.RevokedAt
== null && x.DeliveredAt == null, ct);
            if (pending is not null)
            {
                if (pending.ExpiresAt <= clock.GetUtcNow())
                {
                    await using IDbContextTransaction transaction =
await db.Database.BeginTransactionAsync(ct);
                    pending = await invitations.EnsureAsync(db,
actor.VendorId, user, actor.UserId, "UserInvited", "User", ct);
                    await db.SaveChangesAsync(ct); await
transaction.CommitAsync(ct);
                }
                await DeliverAsync(operation, actor, user, pending, ct);
            }
        }
    }
}

```

```

        return await db.UserAccounts.Where(x => x.Id ==
id).Select(Response).SingleAsync(ct);
    }
    if (await db.UserAccounts.AnyAsync(x => x.Id != id && x.Email == request.Email,
ct))
    { throw new ApiConflictException("email-in-use", "That email cannot be used."); }
    AccountActivation? activation = null;
    await using (IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct))
    {
        await credentials.RevokeCredentialsAsync(db, id, ct, actor.UserId);
        user.Email = request.Email!;
        if (user.Status == "Invited") { activation = await
invitations.EnsureAsync(db, actor.VendorId, user, actor.UserId, "UserInvited", "User", ct); }
        audit.Record(db, actor.UserId, "UserEmailChanged", id);
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
    }
    if (activation is not null) { await DeliverAsync(operation, actor, user,
activation, ct); }
    return await db.UserAccounts.Where(x => x.Id ==
id).Select(Response).SingleAsync(ct);
}

/// <summary>
/// Delivers the account invitation after user creation.
/// </summary>
private async Task DeliverAsync(TenantOperation operation, TenantActor actor, UserAccount
user, AccountActivation activation, CancellationToken ct)
{
    try { await invitations.DeliverAsync(operation.Database, actor.VendorId, user,
activation, ct); }
    catch
    {
        operation.Database.ChangeTracker.Clear();
        audit.Record(operation.Database, actor.UserId,
"UserInvitationDeliveryFailed", user.Id, "Failed");
        await operation.Database.SaveChangesAsync(CancellationToken.None);
        throw;
    }
}
}

```

VersionPublishingRepository.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Vendor;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Repositories.Vendor;

/// <summary>
/// Implements data access operations for version publishing.
/// </summary>
public sealed class VersionPublishingRepository(TenantOperationFactory operations,
ConfigurationArtifactReader reader,

```

```

        ConfigurationValues values, VendorSecurityAudit audit, TimeProvider clock) :
IVersionPublishingRepository
{
    /// <summary>
    /// Validates and publishes a draft version with immutable release evidence.
    /// </summary>
    public async Task<ProductVersionDto> PublishAsync(TenantActor actor, Guid productId, Guid
versionId, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        VendorDbContext db = op.Database;
        ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x => x.Id
== versionId && x.ProductId == productId, ct) ?? throw new ApiAccessException(404);
        List<FileArtifact> files = await db.ProductVersionArtifacts.Where(x =>
x.ProductVersionId == versionId).Select(x => x.FileArtifact).ToListAsync(ct);
        FileArtifact? installer = files.SingleOrDefault(x => x.ArtifactType ==
"Installer"); FileArtifact? schema = files.SingleOrDefault(x => x.ArtifactType == "Schema");
        if (installer is null || schema is null) { throw new
ApiConflictException("release-artifacts-required", "A valid installer and schema are required to
publish."); }

        await reader.VerifyInstallerAsync(op.Tenant.ArchivePath, installer, ct);
        JsonElement schemaJson = await reader.ReadAsync(op.Tenant.ArchivePath, schema,
true, ct);

        if (version.Status == "Published" && version.PublishedAt is not null
&& version.PublishedSchemaArtifactId == schema.Id &&
version.PublishedInstallerArtifactId == installer.Id) { return Response(version); }
        ProductRepository.RequireDraft(version);
        installer.InstallerIdentityJson ??=
System.Text.Json.JsonSerializer.Serialize(await
reader.ReadInstallerIdentityAsync(op.Tenant.ArchivePath, installer, ct));
        version.DeploymentSettingsJson ??= await db.Products.Where(x => x.Id ==
productId).Select(x => x.DeploymentSettingsJson).SingleAsync(ct);
        Guid? upgradeCode =
DeploymentSettingsService.Decode<InstallerIdentity>(installer.InstallerIdentityJson)?.UpgradeCode
;

        if (version.DeploymentSettingsJson is not null && (upgradeCode is null ||
upgradeCode == Guid.Empty))
        { throw new ApiConflictException("upgrade-code-required", "Managed releases
require an MSI UpgradeCode."); }
        ConfigurationRevision? currentDefault = await db.ConfigurationRevisions.Include(x
=> x.FileArtifact)
            .Where(x => x.ProductVersionId == versionId &&
x.ConfigurationProfile.ProductId == productId
            && x.ConfigurationProfile.CustomerId == null &&
x.ConfigurationProfile.InstallationId == null)
            .OrderByDescending(x => x.RevisionNumber).FirstOrDefaultAsync(ct);
        Guid? defaultId = null;
        if (currentDefault is not null)
        {
            try
            {
                JsonElement json = await reader.ReadAsync(op.Tenant.ArchivePath,
currentDefault.FileArtifact, false, ct);
                values.Validate(schemaJson, json, ct); defaultId =
currentDefault.Id;
            }
            catch (Exception e) when (e is ConfigurationValidationException or
ApiConflictException)
            { /* Defaults are optional: freeze absence rather than adding an
undocumented publication gate. */ }
        }
    }
}

```

```

        version.PublishedDefaultRevisionId = defaultId;
        version.PublishedSchemaArtifactId = schema.Id;
version.PublishedInstallerArtifactId = installer.Id;
        version.Status = "Published"; version.PublishedAt = clock.GetUtcNow();
        db.EffectiveConfigurationSignals.Add(new() { ProductVersionId = versionId,
CreatedAt = clock.GetUtcNow() });
        audit.Record(db, actor.UserId, "ProductVersionPublished", versionId,
            currentDefault is not null && defaultId is null ? "PublishedWithoutDefault"
: "Succeeded", "ProductVersion");
        await db.SaveChangesAsync(ct);
        return Response(version);
    }

    /// <summary>
    /// Maps stored entity data to the public response.
    /// </summary>
    private static ProductVersionDto Response(BridgeControl.Models.Vendor.ProductVersion x) =>
new()
    { Id = x.Id, ProductId = x.ProductId, Version = x.Version, Status = x.Status, CreatedAt =
x.CreatedAt, PublishedAt = x.PublishedAt };
}

```

Security/AgentSecurity

AgentActor.cs

```

using System.Security.Claims;

namespace BridgeControl.Security.AgentSecurity;

/// <summary>
/// Represents agent actor data.
/// </summary>
public sealed record AgentActor(Guid VendorId, Guid AgentId, Guid CredentialId)
{
    /// <summary>
    /// Creates an actor identity from the authenticated principal's claims.
    /// </summary>
    public static AgentActor From(ClaimsPrincipal principal) => new(
        Guid.Parse(principal.FindFirst("vendor_id")!.Value),
        Guid.Parse(principal.FindFirst("sub")!.Value),
        Guid.Parse(principal.FindFirst("credential_id")!.Value));
}

```

AgentAuthentication.cs

```

namespace BridgeControl.Security.AgentSecurity;

/// <summary>
/// Defines the agent authentication scheme, policy, and identity claims.
/// </summary>
public static class AgentAuthentication
{
    public const string Scheme = "AgentBearer";
}

```

```

        public const string Policy = "Agent";
    }

```

AgentBearerEvents.cs

```

using System.Security.Claims;
using BridgeControl.Services.AgentServices;
using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace BridgeControl.Security.AgentSecurity;

/// <summary>
/// Rechecks live agent identity after bearer token validation.
/// </summary>
public sealed class AgentBearerEvents(AgentIdentityService identity) : JwtBearerEvents
{
    /// <summary>
    /// Rechecks live account or agent state after bearer token validation.
    /// </summary>
    public override async Task TokenValidated(TokenValidatedContext context)
    {
        ClaimsPrincipal? principal = context.Principal;
        if (principal?.FindFirst("identity_type")?.Value != "agent"
            || !Guid.TryParse(principal.FindFirst("vendor_id")?.Value, out Guid vendor)
            || !Guid.TryParse(principal.FindFirst("sub")?.Value, out Guid agent)
            || !Guid.TryParse(principal.FindFirst("credential_id")?.Value, out Guid
credential)
            || !await identity.ValidateAsync(new(vendor, agent, credential),
context.HttpContext.RequestAborted))
        { context.Fail("Invalid Agent identity."); }
    }
}

```

Security

AgentServiceRegistration.cs

```

using System.Threading.RateLimiting;
using BridgeControl.Options;
using BridgeControl.Repositories.AgentRepositories;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.AgentServices;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Security;

/// <summary>
/// Registers the agent authentication and identity services.
/// </summary>
public static class AgentServiceRegistration
{
    /// <summary>
    /// Registers agent bearer authentication, authorization, and related services.

```

```

    /// </summary>
    public static IServiceCollection AddAgentAuthentication(this IServiceCollection services,
        IConfiguration configuration)
    {
        services.AddOptions<AgentAuthenticationOptions>().Bind(configuration.GetSection(AgentAuthenticati
            onOptions.SectionName))
            .ValidateDataAnnotations().Validate(x => x.HasValidSigningKey(), "Configure
                Authentication:Agent:SigningKey with a Base64 random key of at least 32 bytes.")
            .Validate<IOptions<PlatformAuthenticationOptions>,
                IOptions<TenantAuthenticationOptions>>((agent, platform, tenant) =>
                    agent.HasDistinctSigningKey(platform.Value.SigningKey) &&
                    agent.HasDistinctSigningKey(tenant.Value.SigningKey)
                    && agent.Audience != platform.Value.Audience && agent.Audience !=
                    tenant.Value.Audience,
                    "Agent signing key and audience must differ from both human
                    authentication boundaries.")).ValidateOnStart();
        services.AddScoped<AgentIdentityService>();
        services.AddScoped<IAgentRepository, AgentRepository>();
        services.AddScoped<AgentCredentialOperator>();
        services.AddScoped<AgentBearerEvents>();
        services.AddSingleton<AgentRateLimits>();
        services.AddAuthentication().AddJwtBearer(AgentAuthentication.Scheme, _ => { });

        services.AddOptions<JwtBearerOptions>(AgentAuthentication.Scheme).Configure<IOptions<AgentAuthent
            icationOptions>>((bearer, configured) =>
            {
                AgentAuthenticationOptions options = configured.Value;
                bearer.MapInboundClaims = false; bearer.IncludeErrorDetails = false;
                bearer.SaveToken = false;
                bearer.EventsType = typeof(AgentBearerEvents);
                bearer.TokenValidationParameters = new TokenValidationParameters
                {
                    ValidateIssuer = true,
                    ValidIssuer = options.Issuer,
                    ValidateAudience = true,
                    ValidAudience = options.Audience,
                    ValidateIssuerSigningKey = true,
                    IssuerSigningKey = new
                    SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
                    RequireSignedTokens = true,
                    RequireExpirationTime = true,
                    ValidateLifetime = true,
                    ClockSkew = TimeSpan.Zero,
                    ValidAlgorithms = [SecurityAlgorithms.HmacSha256],
                    ValidTypes = ["at+jwt"],
                    NameClaimType = "sub"
                };
            });
        services.AddAuthorization(options => options.AddPolicy(AgentAuthentication.Policy,
            policy => policy
                .AddAuthenticationSchemes(AgentAuthentication.Scheme).RequireAuthenticatedUser().RequireClaim("id
                    entity_type", "agent")));
        services.AddRateLimiter(options =>
            {
                foreach (string operation in new[] { "AgentRegister", "AgentToken" })
                {
                    options.AddPolicy(operation, context =>
                        RateLimitPartition.GetFixedWindowLimiter($"{operation}:{context.Connection.RemoteIpAddress}", _
                            =>

```



```

        new FixedWindowRateLimiterOptions { PermitLimit = operation
== "AgentToken" ? 60 : 10, Window = TimeSpan.FromMinutes(1), QueueLimit = 0 }));
    }
    });
    return services;
}
}

```

ApiAcceptanceDocumentation.cs

```

using System.Text.Json;
using System.Text.Json.Nodes;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.ApiExplorer;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.OpenApi;

namespace BridgeControl.Security;

/// <summary>Completes generated documentation using the preserved, extracted frozen route
inventory.</summary>
public sealed class ApiAcceptanceDocumentation : IOpenApiDocumentTransformer
{
    /// <summary>
    /// Represents contract operation data.
    /// </summary>
    private sealed record ContractOperation(string Method, string Path, string Summary, string
Authorization);
    /// <summary>
    /// Represents contract data.
    /// </summary>
    private sealed record Contract(ContractOperation[] Operations);

    /// <summary>
    /// Stores the contract operation inventory loaded for API documentation generation.
    /// </summary>
    private static readonly IReadOnlyDictionary<string, ContractOperation> Inventory = Load();

    /// <summary>
    /// Loads approved operation metadata from the embedded API contract.
    /// </summary>
    private static IReadOnlyDictionary<string, ContractOperation> Load()
    {
        using Stream stream =
typeof(ApiAcceptanceDocumentation).Assembly.GetManifestResourceStream("BridgeControl.ApiContract.
json")!;
        return JsonSerializer.Deserialize<Contract>(stream, new JsonSerializerOptions {
PropertyNameCaseInsensitive = true })!.Operations
.ToDictionary(x => x.Method + " " + x.Path.Split('?')[0],
StringComparer.Ordinal);
    }

    /// <summary>
    /// Adds contract metadata and response details to the generated OpenAPI document.
    /// </summary>
    public Task TransformAsync(OpenApiDocument document, OpenApiDocumentTransformerContext
context, CancellationToken cancellationToken)
    {

```

```

foreach (ApiDescription? description in context.DescriptionGroups.SelectMany(x =>
x.Items))
{
    string path = "/" + description.RelativePath;
    if (!document.Paths.TryGetValue(path, out IOpenApiPathItem? item) ||
item.Operations is null || !item.Operations.TryGetValue(new HttpMethod(description.HttpMethod!),
out OpenApiOperation? operation)) { continue; }
    Inventory.TryGetValue(description.HttpMethod + " " + path, out
ContractOperation? contract);
    bool vendorDiscovery = operation.OperationId == "ListLoginVendors";
    if (path.Contains("deployment-settings", StringComparison.Ordinal))
    {
        operation.Summary = description.HttpMethod == "GET" ? "Read
application deployment settings" : operation.OperationId ==
"InitializePublishedDeploymentSettings" ? "Initialize an existing published release once" : "Set
application deployment settings";
        operation.Description = "Application type and local application
paths are managed centrally. Product defaults are copied to versions when published. Published
settings are immutable; initialization is permitted only when missing and is audited. MSI
identity is extracted from verified installer bytes, not supplied by clients.";
    }
    if (operation.OperationId == "AssignProductToAgent")
    { operation.Summary = "Assign a product to an enrolled agent";
operation.Description = "Derives CustomerId from the scoped active agent. Requires
installations.manage and an administrator or scoped technician role. Repeated assignment of the
same agent/product returns the existing active installation. License and desired state are
separate prerequisites for execution. Does not install software immediately." }
    if (operation.OperationId == "ListAgentInstallations")
    { operation.Description = "Returns active owned installations with
application deployment settings and published MSI version mappings. Application is null when
release metadata is incomplete. Agents must not execute missing metadata. Existing installation
identity fields are preserved."; }
    operation.Summary ??= vendorDiscovery ? "List active vendor names for
login" : contract?.Summary ?? "Get the current platform administrator";
    string policy =
description.ActionDescriptor.EndpointMetadata.OfType<IAuthorizeData>().Select(x =>
x.Policy).FirstOrDefault(x => x is not null) ?? "";
    bool anonymous =
description.ActionDescriptor.EndpointMetadata.OfType<IAAllowAnonymous>().Any();
    string permission = contract?.Authorization ?? "PlatformAdmin";
    if (path.Contains("deployment-settings", StringComparison.Ordinal)) {
permission = description.HttpMethod == "GET" ? "products.read" : "products.manage"; }
    if (operation.OperationId == "AssignProductToAgent") { permission =
"installations.manage"; }
    if (permission.Length == 0 && path.Contains("configuration",
StringComparison.Ordinal) && !path.StartsWith("/api/v1/agent/", StringComparison.Ordinal))
    { permission = description.HttpMethod == "GET" ? "configuration.read" :
"configuration.manage"; }
    string authorization = vendorDiscovery ? "Authorization: Anonymous. No
credentials required. Returns only active vendor names, ordered by name, for login and password
recovery. No vendor IDs, accounts or infrastructure details are exposed. Limited to 60 requests
per minute per client IP."
        : anonymous ? "Authorization: " + permission + ". No access bearer
token is required; the documented credential must still be supplied."
        : policy == AgentAuthentication.Policy ? "Authorization:
AgentBearer, current credential and vendor/customer/Agent lifecycle, with ownership of the
requested installation/work."
        : policy == PlatformAuthentication.Policy ? "Authorization:
PlatformBearer for an active platform administrator and session."
        : $"Authorization: TenantBearer, an active session and current
{permission} permissions. Vendor, customer and installation access is checked before reads,
counts and writes; another vendor's identifier never changes the tenant.";
}

```

```

        operation.Description = string.Join("\n\n", new[] { operation.Description,
authorization, Semantics(description.HttpMethod!) }.Where(x => !string.IsNullOrEmpty(x)));
        string tag = path.StartsWith("/api/v1/agent/", StringComparison.Ordinal) ?
"Agent protocol"
        : path.StartsWith("/api/v1/platform/", StringComparison.Ordinal) ?
"Platform"
        : path.Split('/')[3] switch { "auth" => "Tenant authentication",
"agent-enrollments" => "Agents", var resource => char.ToUpperInvariant(resource[0]) +
resource[1..] };
        if (operation.Tags is null || operation.Tags.Count == 0)
        {
            document.Tags ??= new HashSet<OpenApiTag>();
            if (!document.Tags.Any(x => x.Name == tag)) { document.Tags.Add(new
OpenApiTag { Name = tag }); }
            operation.Tags = new HashSet<OpenApiTagReference> { new(tag,
document) };
        }
        operation.Responses ??= [];
        AddProblem(operation, document, "400", "Invalid input, pagination, HTTPS or
origin. Validation errors identify the rejected fields; unknown JSON fields are rejected.");
        if (!anonymous) { AddProblem(operation, document, "401", "Missing, expired,
revoked or wrong-identity credentials."); }
        if (!anonymous && policy == TenantAuthentication.Policy) {
AddProblem(operation, document, "403", "The current account lacks the required permission or
role."); }
        if (path.Contains('{')) { AddProblem(operation, document, "404", "The
resource does not exist or is outside the caller's current scope."); }
        if (!anonymous && description.HttpMethod != "GET") { AddProblem(operation,
document, "409", "The requested change conflicts with current state, ownership or a concurrent
operation."); }
        if (operation.OperationId is "SubmitAgentLogs" or "ReportAgentHeartbeat")
        { operation.Responses.Remove("200"); operation.Responses["204"] = new
OpenApiResponse { Description = "Accepted; no response body. Exact supported retries do not
duplicate state." }; }
        if (operation.OperationId is "SubmitAgentLogs" or "CreateAgentEnrollment"
or "RegisterAgent" or "IssueAgentToken")
        { AddProblem(operation, document, "429", "Rate limit exceeded. Respect the
Retry-After response header."); }
        foreach (OpenApiParameter parameter in (operation.Parameters ??
[]).OfType<OpenApiParameter>())
        {
            if (parameter.Name is "page" or "Page")
            { parameter.Name = "page"; parameter.Description = "One-based page
number; default 1. Pagination overflow is rejected."; if (parameter.Schema is OpenApiSchema
schema) { schema.Default = JsonValue.Create(1); } }
            if (parameter.Name is "pageSize" or "PageSize")
            { parameter.Name = "pageSize"; parameter.Description = "Items per
page: 1-100; default 25. Counts and pages use the same access scope."; if (parameter.Schema is
OpenApiSchema schema) { schema.Default = JsonValue.Create(25); } }
        }
        if (operation.OperationId is "DownloadInstaller" or "DownloadSchema" or
"DownloadAgentArtifact")
        {
            string[] mediaTypes = operation.OperationId == "DownloadInstaller"
? ["application/x-msi"] : operation.OperationId == "DownloadSchema" ? ["application/schema+json"]
: ["application/x-msi", "application/json"];
            operation.Responses["200"] = new OpenApiResponse
            {
                Description = "Verified complete file stream. Filename and
media type come from trusted artifact metadata; internal storage paths are never returned.";
            }
        }
    }
}

```

```

        Content = mediaTypes.ToDictionary(mediaType => mediaType, _
=> new OpenApiMediaType { Schema = new OpenApiSchema { Type = JsonSchemaType.String, Format =
"binary" } })
    };
}
}
return Task.CompletedTask;
}

/// <summary>
/// Describes the contract semantics of the specified HTTP method.
/// </summary>
private static string Semantics(string method) => method switch
{
    "GET" => "Read-only operation. Collection pages default to 25 and are capped at
100 where page/pageSize parameters are listed.",
    "PUT" => "Set/replacement semantics: identical accepted requests do not create
duplicate state. No mandatory If-Match or Idempotency-Key header is introduced.",
    "PATCH" => "Partial update: omitted editable fields are retained. Ownership and
immutable history are not editable. Server concurrency conflicts return 409.",
    "DELETE" => "Removes the documented association, retaining its history. Does not
delete the owning resource.",
    _ => "Creation/command semantics. Retry guarantees are operation-specific; do not
assume every POST can be replayed safely."
};

/// <summary>
/// Adds a documented problem-details response to an OpenAPI operation.
/// </summary>
private static void AddProblem(OpenApiOperation operation, OpenApiDocument document,
string code, string description)
    => operation.Responses!.TryAdd(code, new OpenApiResponse
    {
        Description = description,
        Content = new Dictionary<string, OpenApiMediaType> {
["application/problem+json"] = new() { Schema = new OpenApiSchemaReference("ProblemDetails",
document) } }
    });
}

```

AuthenticationServiceRegistration.cs

```

using System.Threading.RateLimiting;
using BridgeControl.Options;
using BridgeControl.Repositories.Platform;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Security;

/// <summary>
/// Registers shared credential services and the platform's token validation and authorization
rules.
/// </summary>
public static class AuthenticationServiceRegistration

```

```

{
    /// <summary>
    /// Registers platform bearer authentication, authorization, and related services.
    /// </summary>
    public static IServiceCollection AddPlatformAuthentication(this IServiceCollection
services, IConfiguration configuration)
    {
        services.AddSingleton(TimeProvider.System);
        services.AddOptions<PlatformAuthenticationOptions>()
            .Bind(configuration.GetSection(PlatformAuthenticationOptions.SectionName))
            .ValidateDataAnnotations()
            .Validate(options => options.HasValidSigningKey(),
                "Configure Authentication:Platform:SigningKey as a Base64-encoded
random key of at least 32 bytes.")
            .ValidateOnStart();
        services.Configure<PasswordHasherOptions>(options =>
        {
            options.CompatibilityMode = PasswordHasherCompatibilityMode.IdentityV3;
            options.IterationCount = 220_000;
        });
        services.AddSingleton(typeof(IPasswordHasher<>), typeof(PasswordHasher<>));
        services.AddSingleton(typeof>PasswordService<>);
        services.AddSingleton<RefreshCredentialService>();
        services.AddSingleton<AuthenticationSessionService>();
        services.AddSingleton<PlatformAccessTokenService>();
        services.AddScoped<PlatformAuthenticationService>();
        services.AddScoped<IPlatformAccountRepository, PlatformAccountRepository>();
        services.AddScoped<PlatformAdminSetup>();
        services.AddScoped<PlatformAdminRecovery>();
        services.AddScoped<PlatformBearerEvents>();

        services.AddAuthentication(PlatformAuthentication.Scheme)
            .AddJwtBearer(PlatformAuthentication.Scheme, _ => { });
        services.AddOptions<JwtBearerOptions>(PlatformAuthentication.Scheme)
            .Configure<IOptions<PlatformAuthenticationOptions>>((bearer, configured) =>
            {
                PlatformAuthenticationOptions options = configured.Value;
                bearer.MapInboundClaims = false;
                bearer.IncludeErrorDetails = false;
                bearer.SaveToken = false;
                bearer.EventsType = typeof(PlatformBearerEvents);
                bearer.TokenValidationParameters = new TokenValidationParameters
                {
                    ValidateIssuer = true,
                    ValidIssuer = options.Issuer,
                    ValidateAudience = true,
                    ValidAudience = options.Audience,
                    ValidateIssuerSigningKey = true,
                    IssuerSigningKey = new
SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
                    RequireSignedTokens = true,
                    RequireExpirationTime = true,
                    ValidateLifetime = true,
                    ClockSkew = TimeSpan.Zero,
                    ValidAlgorithms = [SecurityAlgorithms.HmacSha256],
                    ValidTypes = ["at+jwt"],
                    NameClaimType = "sub",
                    RoleClaimType = "role"
                };
            });
        services.AddAuthorization(options =>
        {

```

```

        options.FallbackPolicy = new
AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();
        options.AddPolicy(PlatformAuthentication.Policy, policy => policy

.AddAuthenticationSchemes(PlatformAuthentication.Scheme).RequireAuthenticatedUser()
        .RequireClaim(PlatformAuthentication.IdentityTypeClaim,
PlatformAuthentication.IdentityType)
        .RequireRole(PlatformAuthentication.Role));
    });
    services.AddRateLimiter(options =>
    {
        options.RejectionStatusCode = StatusCodes.Status429TooManyRequests;
        options.AddPolicy(PlatformAuthentication.RateLimitPolicy, context =>
        {
            // Route names keep case variants and trailing slashes in the same
rate-limit bucket.
            bool refresh =
context.GetEndpoint()?.Metadata.GetMetadata<IEndpointNameMetadata>()?.EndpointName ==
"PlatformRefresh";
            string operation = refresh ? "refresh" : "login";
            PlatformAuthenticationOptions limits =
context.RequestServices.GetRequiredService<IOptions<PlatformAuthenticationOptions>>().Value;
            return RateLimitPartition.GetFixedWindowLimiter(
                $"{operation}:{context.Connection.RemoteIpAddress}", _ =>
new FixedWindowRateLimiterOptions
            {
                PermitLimit = refresh ?
limits.RefreshRequestsPerMinute : limits.LoginRequestsPerMinute,
                Window = TimeSpan.FromMinutes(1),
                QueueLimit = 0,
                AutoReplenishment = true
            });
        });
        options.OnRejected = async (context, cancellationToken) =>
        {
            if (context.Lease.TryGetMetadata(MetadataName.RetryAfter, out
TimeSpan retryAfter))
            {
                context.HttpContext.Response.Headers.RetryAfter =
Math.Ceiling(retryAfter.TotalSeconds).ToString(System.Globalization.CultureInfo.InvariantCulture);
            }
            await Results.Problem(statusCode:
StatusCodes.Status429TooManyRequests,
                title: "Too many authentication attempts. Try again
later.").ExecuteAsync(context.HttpContext);
        };
    });
    return services;
}
}

```

HostingProfile.cs

```

using System.Net;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.KeyManagement;
using Microsoft.AspNetCore.DataProtection.Repositories;

```

```

using Microsoft.AspNetCore.DataProtection.XmlEncryption;
using Microsoft.AspNetCore.HttpOverrides;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Options;

namespace BridgeControl.Security;

/// <summary>Explicit internet origin, one trusted proxy hop, and persistent service-identity
protected keys.</summary>
public static class HostingProfile
{
    /// <summary>
    /// Registers and validates hosting, proxy, and transport-security settings.
    /// </summary>
    public static void AddHostingProfile(this IServiceCollection services, IConfiguration
configuration, IHostEnvironment environment)
    {
        services.AddOptions<HostingProfileOptions>().Bind(configuration.GetSection("HostingProfile"))
            .Validate(x => Valid(x, environment.IsDevelopment()), "Configure an HTTPS
public origin, explicit proxy IP addresses and a private key directory for production.")
            .Validate(_ => environment.IsDevelopment() ||
SecureSql(configuration.GetConnectionString("PlatformDatabase"))
            && SecureSql(configuration.GetConnectionString("VendorServer")) ??
configuration.GetConnectionString("PlatformDatabase"), "Production SQL connections must require
encryption and certificate validation.").ValidateOnStart();

        services.AddOptions<ForwardedHeadersOptions>().Configure<Options<HostingProfileOptions>>((forwar
ding, profile) =>
        {
            forwarding.ForwardedHeaders = ForwardedHeaders.XForwardedFor |
ForwardedHeaders.XForwardedProto;
            forwarding.ForwardLimit = 1; forwarding.RequireHeaderSymmetry = true;
            forwarding.KnownProxies.Clear(); forwarding.KnownIPNetworks.Clear();
            // ASP.NET treats both empty allowlists as trust-all. A sentinel prevents
that in direct TLS mode.
            if (profile.Value.TrustedProxies.Length == 0) {
forwarding.KnownProxies.Add(IPAddress.None); }
            foreach (string value in profile.Value.TrustedProxies) {
forwarding.KnownProxies.Add(IPAddress.Parse(value)); }
        });
        services.AddDataProtection().SetApplicationName("BridgeControl");

        services.AddOptions<KeyManagementOptions>().Configure<Options<HostingProfileOptions>,
ILoggerFactory>((keys, profile, logger) =>
        {
            if (string.IsNullOrEmpty(profile.Value.DataProtectionKeyDirectory)) {
return; }

            if (!OperatingSystem.IsWindows()) { throw new
InvalidOperationException("The supported production key store requires Windows DPAPI."); }
            keys.XmlRepository = new FileSystemXmlRepository(new
DirectoryInfo(PrivateDirectory.Ensure(profile.Value.DataProtectionKeyDirectory)), logger);
            keys.XmlEncryptor = new DpapiXmlEncryptor(protectToLocalMachine: false,
logger);
        });
    }

    /// <summary>
    /// Checks that hosting settings satisfy the requirements of the selected environment.
    /// </summary>
    public static bool Valid(HostingProfileOptions profile, bool development)
    {

```

```

        if (profile.TrustedProxies.Any(x => !IPAddress.TryParse(x, out IPAddress? ip) ||
ip.Equals(IPAddress.Any) || ip.Equals(IPAddress.IPv6Any) || ip.Equals(IPAddress.None))) {return
false; }

        if (profile.RequireProxy && profile.TrustedProxies.Length == 0) { return false; }
        bool origin = Uri.TryCreate(profile.PublicOrigin, UriKind.Absolute, out Uri? uri)
&& uri.Scheme == "https" && uri.AbsolutePath == "/"
        && uri.Query.Length == 0 && uri.Fragment.Length == 0 && uri.UserInfo.Length
== 0;

        return (development && profile.PublicOrigin.Length == 0 || origin) &&
Path.IsPathFullyQualified(profile.LogDirectory)
        && (development || OperatingSystem.IsWindows()) &&
Path.IsPathFullyQualified(profile.DataProtectionKeyDirectory));
    }

    /// <summary>
    /// Checks that a SQL Server connection uses the required transport-security settings.
    /// </summary>
    public static bool SecureSql(string? connection)
    {
        try
        {
            SqlConnectionStringBuilder settings = new(connection);
            return !string.IsNullOrEmpty(settings.DataSource) &&
!settings.TrustServerCertificate
            && settings.Encrypt != SqlConnectionEncryptOption.Optional &&
!settings.DataSource.StartsWith("(localdb)", StringComparison.OrdinalIgnoreCase);
        }
        catch (ArgumentException) { return false; }
    }

    /// <summary>
    /// Adds the configured proxy and request-security middleware.
    /// </summary>
    public static void UseHostingProfile(this WebApplication app)
    {
        app.Use(async (context, next) =>
        {
            HostingProfileOptions profile =
context.RequestServices.GetRequiredService<Options<HostingProfileOptions>>().Value;
            if (profile.RequireProxy && !profile.TrustedProxies.Any(x =>
IPAddress.Parse(x).MapToIPv6().Equals(context.Connection.RemoteIpAddress?.MapToIPv6())))
            { context.Response.StatusCode = StatusCodes.Status400BadRequest; return; }
            await next(context);
        });
        app.UseForwardedHeaders();
        app.Use(async (context, next) =>
        {
            HostingProfileOptions profile =
context.RequestServices.GetRequiredService<Options<HostingProfileOptions>>().Value;
            if (!string.IsNullOrEmpty(profile.PublicOrigin))
            {
                Uri origin = new(profile.PublicOrigin);
                bool hostMatches = string.Equals(context.Request.Host.Host,
origin.Host, StringComparison.OrdinalIgnoreCase)
                && (context.Request.Host.Port ?? 443) == origin.Port;
                bool unsafeMethod = !HttpMethods.IsGet(context.Request.Method) &&
!HttpMethods.IsHead(context.Request.Method) && !HttpMethods.IsOptions(context.Request.Method);
                bool originMatches = !context.Request.Headers.ContainsKey("Origin")
|| string.Equals(context.Request.Headers.Origin.ToString(),
origin.GetLeftPart(UriPartial.Authority), StringComparison.OrdinalIgnoreCase);
                if (!hostMatches || unsafeMethod && !originMatches)

```



```

        { await Results.Problem(statusCode:
StatusCodes.Status400BadRequest, title: "Request origin is not
permitted.").ExecuteAsync(context); return; }
    }
    await next(context);
});
}
}
}

```

PlatformAuthentication.cs

```

namespace BridgeControl.Security;

/// <summary>
/// Names the platform authentication boundary and claims used by protected platform endpoints.
/// </summary>
public static class PlatformAuthentication
{
    public const string Scheme = "PlatformBearer";
    public const string Policy = "PlatformAdministrator";
    public const string Role = "PlatformAdmin";
    public const string IdentityTypeClaim = "identity_type";
    public const string IdentityType = "platform_admin";
    public const string SessionIdClaim = "sid";
    public const string ActiveStatus = "Active";
    public const string RateLimitPolicy = "PlatformAuthentication";
}

```

PlatformBearerEvents.cs

```

using System.IdentityModel.Tokens.Jwt;
using BridgeControl.Services.Platform;
using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace BridgeControl.Security;

/// <summary>
/// Rejects platform tokens whose identity type, account or server-side session is invalid.
/// </summary>
public class PlatformBearerEvents(PlatformAuthenticationService authentication) : JwtBearerEvents
{
    /// <summary>
    /// Rechecks live account or agent state after bearer token validation.
    /// </summary>
    public override async Task TokenValidated(TokenValidatedContext context)
    {
        if (context.Principal?.FindFirst(PlatformAuthentication.IdentityTypeClaim)?.Value
!= PlatformAuthentication.IdentityType
        ||
!Guid.TryParse(context.Principal.FindFirst(JwtRegisteredClaimNames.Sub)?.Value, out Guid
accountId)
        ||
!Guid.TryParse(context.Principal.FindFirst(PlatformAuthentication.SessionIdClaim)?.Value, out
Guid sessionId)
        || !await authentication.ValidateSessionAsync(sessionId, accountId,
context.HttpContext.RequestAborted))
        {

```

```

        context.Fail("The platform session is invalid.");
    }
}

```

PlatformOpenApiTransformer.cs

```

using System.Text.Json.Nodes;
using BridgeControl.Security.AgentSecurity;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.OpenApi;
using Microsoft.OpenApi;

namespace BridgeControl.Security;

/// <summary>
/// Documents platform bearer authentication and the refresh cookie/header contract in OpenAPI.
/// </summary>
public class PlatformOpenApiTransformer : IOpenApiDocumentTransformer
{
    /// <summary>
    /// Documents platform bearer authentication in the generated OpenAPI document.
    /// </summary>
    public Task TransformAsync(OpenApiDocument document, OpenApiDocumentTransformerContext
context, CancellationToken cancellationToken)
    {
        document.Info.Title = "BridgeControl API";
        document.Info.Version = "v1";
        document.Info.Description = "Implemented BridgeControl operations. Use HTTPS.
Login sets the HTTP-only refresh cookie; authorize protected requests with the returned access
token. The API specification remains the controlling contract.";
        document.Components ??= new OpenApiComponents();
        document.Components.SecuritySchemes ??= new Dictionary<string,
IOpenApiSecurityScheme>();
        document.Components.SecuritySchemes[PlatformAuthentication.Scheme] = new
OpenApiSecurityScheme
        {
            Type = SecuritySchemeType.Http,
            Scheme = "bearer",
            BearerFormat = "JWT",
            Description = "Platform access token returned by login or refresh."
        };
        document.Components.SecuritySchemes["PlatformRefreshCookie"] = new
OpenApiSecurityScheme
        {
            Type = SecuritySchemeType.ApiKey,
            In = ParameterLocation.Cookie,
            Name = PlatformRefreshCookie.Name,
            Description = "HTTP-only cookie set by platform login and rotated on
refresh."
        };
        document.Components.SecuritySchemes[TenantAuthentication.Scheme] = new
OpenApiSecurityScheme
        { Type = SecuritySchemeType.Http, Scheme = "bearer", BearerFormat = "JWT",
Description = "Vendor/customer access token. Current session and grants are checked on each
request." };
        document.Components.SecuritySchemes[AgentAuthentication.Scheme] = new
OpenApiSecurityScheme

```

```

        { Type = SecuritySchemeType.Http, Scheme = "bearer", BearerFormat = "JWT",
Description = "Agent-only access token. Current credential, Agent, customer and vendor lifecycle
are checked on every request." };
        document.Components.SecuritySchemes["TenantRefreshCookie"] = new
OpenApiSecurityScheme
        { Type = SecuritySchemeType.ApiKey, In = ParameterLocation.Cookie, Name =
TenantRefreshCookie.Name, Description = "Separate secure tenant refresh cookie, rotated on use."
};

        foreach (Microsoft.AspNetCore.Mvc.ApiExplorer.ApiDescription description in
context.DescriptionGroups.SelectMany(group => group.Items))
        {
            string path = "/" + description.RelativePath;
            if (!document.Paths.TryGetValue(path, out IOpenApiPathItem? item) ||
item.Operations is null
                || !item.Operations.TryGetValue(new
HttpMethod(description.HttpMethod!), out OpenApiOperation? operation))
            {
                continue;
            }
            (operation.Summary, operation.Description) = path switch
            {
                "/api/v1/agent/logs" => ("Submit Agent logs", "Accepts 1-100
entries, messages up to 2000 characters and a 256 KiB request. Installation ownership is checked
for every entry; sensitive content is removed. Exact entries deduplicate within retention. 10
batches/500 entries per Agent per minute; storage quota returns 409."),
                "/api/v1/logs" => ("Search scoped logs", "Requires logs.read.
Bounded pages, exact severity and inclusive-from/exclusive-to timestamp filters.
Installation-only viewers cannot see unbound Agent logs. Sensitive entries are restricted to
vendor administrators after redaction."),
                "/api/v1/installations/{installationId}/logs" => ("Read
installation logs", "Current logs.read installation grants are applied before filters, counts and
pagination."),
                "/api/v1/agents/{agentId}/logs" => ("Read accessible Agent logs",
"Seeing an Agent does not grant visibility of its other installations or Agent-wide messages."),
                "/api/v1/monitoring/overview" => ("Read operational health",
"Requires monitoring.read. Scoped totals and a bounded installation page use live lifecycle,
license, accepted state and work records. Online is within 5 minutes; stale is 5-15 minutes.
Missing state remains unknown. No raw Agent/configuration error text is exposed."),
                "/api/v1/deployments" => ("Search deployments", "Paged deployment
history scoped by current deployments.read grants before filters and counts. Optional
installationId, customerId and status filters."),
                "/api/v1/deployments/{deploymentId}" => ("Read a deployment and its
attempts", "Includes ordered attempt history and safe error summaries. Command success is
separate from actual-state reconciliation."),
                "/api/v1/installations/{installationId}/deployments" => ("Read
installation deployment history", "Paged history within the caller's current installation
scope."),
                "/api/v1/deployments/{deploymentId}/cancel" => ("Cancel pending
deployment", "Requires deployments.manage. Repeated cancellation is idempotent; running and
completed operations return 409. Cancellation fences this desired revision."),
                "/api/v1/agent/deployments" => ("Retrieve eligible pending work",
"Agent-owned candidate pages default to 25 and allow at most 100. Follow the next Link even when
an eligibility-filtered page is empty; restart at page 1 on each poll. Targets, licenses and
active attempts are rechecked. Retrieval does not start execution."),
                "/api/v1/agent/deployments/{deploymentId}/attempts" => ("Start or
recover a deployment attempt", "Returns the same running attempt on retries while still eligible.
At most three failed attempts, with 30/60-second backoff. Persist the returned identity before
execution. No client retry key is required."),
                "/api/v1/agent/deployments/{deploymentId}/attempts/{attemptId}" =>
("Report deployment progress or outcome", "Running, Installing, Configuring, then Succeeded,
Failed or RestartRequired. Terminal reports require CompletedAt. Exact repeats are idempotent;
completed history is immutable. Untrusted errors are replaced with safe summaries."),
            }

```

```

        "/api/v1/platform/auth/login" => ("Sign in as a platform
administrator", "Verifies the exact password, creates a revocable session and returns an access
token. The browser receives a Secure, HTTP-only refresh cookie. Invalid credentials or an
inactive account return the same generic 401."),
        "/api/v1/platform/auth/refresh" => ("Renew the platform session",
"Requires the refresh cookie from login and X-BridgeControl-Refresh: 1. Rotates the cookie and
returns a new access token. Reusing an already consumed refresh credential revokes the session.
Coordinate refresh so only one request runs per session."),
        "/api/v1/platform/auth/logout" => ("Sign out of the current
platform session", "Requires a platform bearer token. Revokes this session immediately and clears
its refresh cookie. Other sessions are independent."),
        "/api/v1/platform/account/me" => ("Get the current platform
administrator", "Documented in section 4.1: returns the administrator for an active platform
session without exposing credential hashes or infrastructure secrets."),
        _ => (operation.Summary, operation.Description)
    };
    operation.Responses ??= [];
    if (operation.OperationId is "UploadInstaller" or "UploadSchema")
    {
        operation.Description = "Upload one multipart field named file.
Checksums and content type are determined by the server. Draft versions only; complete validation
precedes attachment. See ProductAdministration.md and decision 0046 for limits and supported
formats.";

        operation.RequestBody = new OpenApiRequestBody
        {
            Required = true,
            Content = new Dictionary<string, OpenApiMediaType>
            {
                ["multipart/form-data"] = new()
                {
                    Schema = new OpenApiSchema
                    {
                        Type = JsonSchemaType.Object,
                        Required = new HashSet<string> {
"file" },
                        Properties = new Dictionary<string,
IOpenApiSchema> { ["file"] = new OpenApiSchema { Type = JsonSchemaType.String, Format = "binary"
} }
                    }
                }
            }
        };
        (operation.Summary, operation.Description) = operation.OperationId switch
        {
            "GetDesiredState" or "GetReportedState" => ("Read scoped
installation state", "Requires installations.read and current installation scope. Desired and
accepted actual state are independent; missing state returns 404."),
            "SetDesiredState" => ("Set a licensed published installation
target", "Requires installations.manage and current eligibility. The effective revision must
match the installation/version and current validated source state. Identical PUTs reuse the
target; changes atomically schedule reconciliation and supersede old pending work."),
            "GetAgentDesiredState" => ("Retrieve the current authorized
target", "Rechecks Agent identity, binding, lifecycle, licensing and immutable artifacts. The
Link response header identifies authorized installer and configuration downloads. Refresh after
reconnect and before new work."),
            "ReportInstallationState" => ("Report an ordered actual
installation observation", "Accepts Running, Ready, Stopped, NotInstalled, Unknown, Failed or
RestartRequired. Ready means a desktop application is installed with its configuration on disk,
without requiring a running user session; installed version, configuration revision and hash are
required. ReportedAt must not exceed server time by more than five minutes. Identical retries

```

```

deduplicate; conflicting or older reports return 409. License expiry does not prevent
reporting."),
    "DownloadAgentArtifact" => ("Download an artifact referenced by
current authorized desired state", "Verifies complete archive size/hash before streaming, returns
ETag/Content-Digest and version/revision metadata, and serves complete responses without range
processing. Superseded, unrelated or unlicensed references do not authorize download."),
    "PreviewConfigurationMigration" => ("Review a configuration
migration", "Requires configuration.manage and an active scoped installation. Returns candidate
values, validity, validation errors and changes. This replaces the user's latest review for this
installation/target; it does not change current configuration."),
    "ApplyConfigurationMigration" => ("Apply the user's latest reviewed
migration", "Uses the current user's latest preview for this installation and target. Rejects
expired, unresolved or changed reviews and requires exactly the reported destructive rule IDs.
Successful retries return the original effective revision. No preview identifier is accepted."),
    "ListConfigurationMigrations" or "GetConfigurationMigration" =>
("Read version migration definitions", "Requires configuration.read in the active vendor. Rules
are returned in sequence order; lists use bounded pagination."),
    "CreateConfigurationMigration" or "ReplaceConfigurationMigration"
=> ("Define ordered configuration transformations", "Requires a Vendor Administrator with
configuration.manage. Both versions must belong to this product; replacements retain the version
pair and archive immutable rule revisions. Cycles, duplicate edges and unsupported
transformations are rejected."),
    "GetEffectiveConfiguration" => ("Read the current generated
configuration", "Requires configuration.read and live installation scope. Recalculation is
asynchronous; failed generation preserves the prior valid revision. An existing target version is
not switched by another version's source changes."),
    "ListEffectiveConfigurationRevisions" or
"GetEffectiveConfigurationRevision" => ("Read immutable generated configuration history",
"Returns exact archived values, version and source revision IDs. Revision lookup remains scoped
to its installation. History uses bounded pagination in revision-number order; configuration
remains read-only."),
    "ListInstallations" or "GetInstallation" => ("Read accessible
installations", "Uses current customer/technician/installation assignments. Collection
customerId, productId and status filters apply before counts and pagination. Status is lifecycle
Active or Disabled."),
    "UpdateInstallation" => ("Rename an installation", "PATCH accepts
only a supplied name. Ownership, registration identity and telemetry remain unchanged. Agent
registration retries continue to use the original registration name."),
    "DeactivateInstallation" => ("Deactivate an installation", "Stops
new Agent work and releases active license capacity while retaining assignments and history. The
Agent may continue managing its other installations."),
    "ReactivateInstallation" => ("Reactivate an installation",
"Requires available ownership and, if a license remains assigned, current validity and available
capacity. Returns 409 when capacity has been consumed elsewhere."),
    "ListLicenses" or "GetLicense" => ("Read scoped licenses",
"Requires licenses.read. Customer users see only licenses currently assigned to their accessible
installations. Lists support customerId/productId/status and bounded pagination."),
    "CreateLicense" => ("Create a customer/product license", "Requires
licenses.manage and an active in-scope customer/product. ValidUntil must be later than ValidFrom;
null expiry or capacity means unlimited in that dimension. Capacity, when present, must be
positive."),
    "UpdateLicense" => ("Edit license terms", "Ownership and lifecycle
cannot be patched. Omitted terms remain unchanged; explicit null clears ValidUntil or
MaxInstallations. Reducing capacity below active assignments returns 409."),
    "DeactivateLicense" or "ReactivateLicense" => ("Change license
lifecycle", "Preserves assignments/history. Disabled or expired licenses never authorize new
deployment work; reactivation does not alter validity dates."),
    "AssignInstallationLicense" => ("Assign or replace an installation
license", "Requires licenses.manage on both resources, matching customer/product, current
validity and capacity. Assignment is atomic with history and audit; valid identical retries
return 204."),

```

```

        "RemoveInstallationLicense" => ("Remove an installation license",
"Returns 204, including an already-unassigned installation. Retains assignment history and
immediately removes eligibility for new deployment work."),
        "CreateAgentEnrollment" => ("Create a single-use Agent enrollment",
"Requires agents.manage and customer scope. Returns a secret enrollmentCode once, valid for 15
minutes. Limited to 10 requests per minute per authorized account."),
        "RegisterAgent" => ("Exchange an enrollment for an Agent identity",
"Atomically consumes enrollmentCode and returns agentId, credential and expiresAt once. Replays
return 401. A lost response requires a new authorized enrollment. Limited to 10/minute/IP and
enrollment."),
        "IssueAgentToken" => ("Authenticate an Agent credential", "Accepts
credential and returns accessToken and expiresAt. Separate Agent bearer identity, valid for at
most 10 minutes and never beyond credential expiry. Limited to 60/minute/IP and credential."),
        "ListAgents" or "GetAgent" or "ListAgentHeartbeats" => ("Read
scoped Agent status", "Requires agents.read. Customer and technician scope applies; customer
users need an assigned installation managed by the Agent. Heartbeat history uses bounded
pagination in receipt order."),
        "ListAgentInstallations" => ("List this Agent's active
installations", "Derives vendor, customer and Agent from the current machine identity. Returns a
bounded page; disabled installations are excluded."),
        "RegisterAgentInstallation" => ("Register an Agent-managed
installation", "Customer + Product + trimmed Name identifies a logical installation. Retry by its
owning Agent returns the existing row. Another Agent or unassigned legacy installation conflicts;
ownership is never transferred."),
        "ReportAgentHeartbeat" => ("Report Agent status", "ReportedAt
identifies a heartbeat. Exact retries return 204; changed content at that timestamp returns 409.
Older reports remain in history without replacing current status. Default timestamps and reports
over five minutes ahead are rejected."),
        "PublishProductVersion" => ("Publish a product version", "Requires
verified MSI and schema artifacts. Pins immutable release references and an optional compatible
default; no default is required. Identical retries return the same publication timestamp."),
        "SetProductConfiguration" => ("Set product defaults", "Vendor
administrators only. Validates normalized values against the selected product version's schema.
Creates an immutable revision, or reuses identical current values for that version. Published
defaults cannot change."),
        "SetCustomerConfiguration" => ("Set a customer/product override",
"Checks current customer scope and selected product/version ownership. PUT replaces this scope's
values; omitted required schema properties still fail validation."),
        "SetInstallationConfiguration" => ("Set installation overrides",
"Derives product/customer ownership from the authorized installation. Values must validate
against that product version's schema. Changes enqueue durable recalculation notifications."),
        "GetProductConfiguration" or "GetCustomerConfiguration" or
"GetInstallationConfiguration" => ("Get current source configuration", "Returns the newest
revision in this profile, including ProductVersionId and exact stored values. An absent profile
returns 404."),
        "ListProductConfigurationRevisions" or
"ListCustomerConfigurationRevisions" or "ListInstallationConfigurationRevisions" => ("List source
configuration history", "Paged immutable history, newest first. Counts and records use the same
authorized profile. Storage and internal schema references are not exposed."),
        "ListPlatformVendors" => ("List vendors", "Platform administrators
only. Returns a bounded page without database or archive references."),
        "GetPlatformVendor" => ("Get a vendor", "Platform administrators
only. Status reflects provisioning, awaiting an initial administrator, or the vendor
lifecycle."),
        "CreatePlatformVendor" => ("Create and provision a vendor",
"Creates its isolated database and archive on demand. Retry the same name after an interrupted
setup to resume it. Infrastructure-ready vendors await their initial administrator invitation."),
        "UpdatePlatformVendor" => ("Edit a vendor name", "Only a supplied
name is changed. Names are trimmed and unique case-insensitively; internal identity is unchanged.
Lifecycle changes use separate documented endpoints."),
        "InviteInitialVendorAdministrator" => ("Invite the initial vendor
administrator", "Creates an invited account with the predefined vendor administrator role.

```

Development delivery writes a private local email file. The response never contains an activation token or password. Repeat the same email to retry delivery.),

"ActivateInvitedAccount" => ("Activate an invited account", "Submit the private single-use activation token and choose a 10-128 character password. Whitespace is preserved. Successful activation returns 204 and does not sign the user in."),

"ListUsers" => ("List accessible users", "Filters and pagination apply after vendor/customer scope. Customer administrators only see their own customer."),

"InviteUser" => ("Invite a scoped user", "Accepts email, fixed customer ownership and compatible predefined roles. The invited user chooses a password through activation. Repeat an identical pending invitation to retry private delivery."),

"UpdateUser" => ("Change a user email", "Only a supplied email changes. Ownership, passwords, roles and lifecycle fields are not editable here. An email change revokes sessions and pending credentials; invited accounts receive a replacement invitation."),

"ListCustomers" => ("List accessible customers", "Technicians see only assigned customers. Counts, filters and pagination use the same scope."),

"CreateCustomer" => ("Create a customer", "Vendor administration only. Returns the documented customer fields."),

"UpdateCustomer" => ("Rename a customer", "Changes a supplied name only. Omitted fields are preserved and concurrent server edits return 409."),

"DeactivateCustomer" => ("Deactivate a customer", "Revokes customer user sessions and pending credentials atomically with audit evidence, retaining records and history."),

"ReactivateCustomer" => ("Reactivate a customer", "Restores customer availability without restoring old sessions or credentials."),

"ListAssignableRoles" => ("List assignable roles", "Predefined roles within the administrator's grantable scope; no custom-role CRUD."),

"ListKnownPermissions" => ("List known scoped permissions", "Returns the permission catalog associated with assignable roles."),

"SetUserRoles" => ("Replace a user's roles", "Idempotent set replacement. Customer ownership is fixed; mixed vendor/customer scopes are rejected. Role reductions apply to existing sessions."),

"GetCustomerAccess" => ("Get technician customer assignments", "Vendor administrators only. Returns the current customerIds set."),

"SetCustomerAccess" => ("Replace technician customer assignments", "Vendor administrators only. All IDs must be distinct and belong to the current vendor. Removal immediately limits access."),

"GetInstallationAccess" => ("Get customer-user installation assignments", "Scoped administrators only. Returns the current installationIds set."),

"SetInstallationAccess" => ("Replace customer-user installation assignments", "Every installation must belong to the subject user's fixed customer. Ordinary users cannot delegate access. Identical replacements create no duplicate grants."),

_ => (operation.Summary, operation.Description)

};

operation.Responses.TryAdd("500", new OpenApiResponse

{

Description = "Unexpected server failure. The response contains a trace ID and no internal exception details.",

Content = new Dictionary<string, OpenApiMediaType>

{

["application/problem+json"] = new() { Schema = new OpenApiSchemaReference("ProblemDetails", document) }

}

});

bool anonymous =

description.ActionDescriptor.EndpointMetadata.OfType<IAAllowAnonymous>().Any();

if (!anonymous &&

description.ActionDescriptor.EndpointMetadata.OfType<IAuthorizeData>().

.Any(metadata => metadata.Policy == PlatformAuthentication.Policy))

{

operation.Security = [new OpenApiSecurityRequirement

{

```

                                [new
OpenApiSecuritySchemeReference(PlatformAuthentication.Scheme, document)] = []
                                });
                                }
                                if (!anonymous &&
description.ActionDescriptor.EndpointMetadata.OfType<IAuthorizeData>().Any(metadata =>
metadata.Policy == TenantAuthentication.Policy))
                                { operation.Security = [new OpenApiSecurityRequirement { [new
OpenApiSecuritySchemeReference(TenantAuthentication.Scheme, document)] = [] }]; }
                                if (!anonymous &&
description.ActionDescriptor.EndpointMetadata.OfType<IAuthorizeData>().Any(metadata =>
metadata.Policy == AgentAuthentication.Policy))
                                { operation.Security = [new OpenApiSecurityRequirement { [new
OpenApiSecuritySchemeReference(AgentAuthentication.Scheme, document)] = [] }]; }
                                if (path == "/api/v1/platform/auth/refresh" || path ==
"/api/v1/auth/refresh")
                                {
                                    operation.Security = [new OpenApiSecurityRequirement
                                    {
                                        [new OpenApiSecuritySchemeReference(path ==
"/api/v1/auth/refresh" ? "TenantRefreshCookie" : "PlatformRefreshCookie", document)] = []
                                    }];
                                    operation.Parameters ??= [];
                                    operation.Parameters.Add(new OpenApiParameter
                                    {
                                        Name = PlatformRefreshCookie.RequestHeader,
                                        In = ParameterLocation.Header,
                                        Required = true,
                                        Description = "Send the value 1. Browser clients must
include the refresh cookie.",
                                        Schema = new OpenApiSchema { Type = JsonSchemaType.String,
Default = JsonValue.Create("1"), Enum = [JsonValue.Create("1")!] }
                                    });
                                }
                                if (document.Components.Schemas?.TryGetValue("PlatformLoginRequest", out
IOpenApiSchema? login) == true
                                && login.Properties?.TryGetValue("password", out IOpenApiSchema? password)
== true && password is OpenApiSchema passwordSchema)
                                {
                                    passwordSchema.Format = "password";
                                    passwordSchema.WriteOnly = true;
                                    passwordSchema.Description = "Exact password; whitespace is preserved. This
is never returned in a response.";
                                }
                                if (document.Components.Schemas?.TryGetValue("ActivateAccountRequest", out
IOpenApiSchema? activation) == true)
                                {
                                    foreach (string field in new[] { "token", "password" })
                                    {
                                        if (activation.Properties?.TryGetValue(field, out IOpenApiSchema?
property) == true && property is OpenApiSchema secret)
                                        {
                                            secret.Format = "password";
                                            secret.WriteOnly = true;
                                        }
                                    }
                                }
                                foreach (IOpenApiSchema schema in document.Components.Schemas?.Values ?? [])
                                {
                                    foreach (KeyValuePair<string, IOpenApiSchema> field in schema.Properties ??
new Dictionary<string, IOpenApiSchema>())

```



```

        {
            if (field.Key is "password" or "currentPassword" or "newPassword"
or "token" && field.Value is OpenApiSchema secret)
            { secret.Format = "password"; secret.WriteOnly = true; }
        }
    }
    return Task.CompletedTask;
}
}

```

PlatformRefreshCookie.cs

```

namespace BridgeControl.Security;

/// <summary>
/// Writes and clears the platform refresh credential using the same secure cookie settings.
/// </summary>
public static class PlatformRefreshCookie
{
    public const string Name = "__Secure-BridgeControl.Platform.Refresh";
    public const string RequestHeader = "X-BridgeControl-Refresh";
    private const string Path = "/api/v1/platform/auth";

    /// <summary>
    /// Writes the refresh credential in a secure, HTTP-only cookie with its expiry.
    /// </summary>
    public static void Write(HttpResponse response, string credential, DateTimeOffset expiry)
    {
        CookieOptions options = CreateOptions();
        options.Expires = expiry;
        response.Cookies.Append(Name, credential, options);
    }

    /// <summary>
    /// Deletes the refresh credential cookie using its original path and security settings.
    /// </summary>
    public static void Clear(HttpResponse response) => response.Cookies.Delete(Name,
CreateOptions());

    /// <summary>
    /// Creates the security and path settings for the refresh cookie.
    /// </summary>
    private static CookieOptions CreateOptions() => new()
    {
        HttpOnly = true,
        Secure = true,
        SameSite = SameSiteMode.Strict,
        Path = Path,
        IsEssential = true
    };
}

```

TenantAuthentication.cs

```

using System.Security.Claims;
using BridgeControl.Services;
using BridgeControl.Services.Vendor;

```

```

using Microsoft.AspNetCore.Authentication.JwtBearer;

namespace BridgeControl.Security;

/// <summary>
/// Defines tenant authentication constants and reads tenant identity claims.
/// </summary>
public static class TenantAuthentication
{
    public const string Scheme = "TenantBearer";
    public const string Policy = "TenantUser";

    /// <summary>
    /// Reads the authenticated tenant user's identifier from the subject claim.
    /// </summary>
    public static Guid UserId(ClaimsPrincipal user) =>
        Guid.Parse(user.FindFirst("sub")!.Value);

    /// <summary>
    /// Reads the authenticated vendor identifier from the principal's claims.
    /// </summary>
    public static Guid VendorId(ClaimsPrincipal user) =>
        Guid.Parse(user.FindFirst("vendor_id")!.Value);

    /// <summary>
    /// Reads the tenant authentication session identifier from the principal's claims.
    /// </summary>
    public static Guid SessionId(ClaimsPrincipal user) =>
        Guid.Parse(user.FindFirst("sid")!.Value);
}

/// <summary>
/// Rechecks live tenant account and session state after token validation.
/// </summary>
public class TenantBearerEvents(TenantAuthenticationService authentication) : JwtBearerEvents
{
    /// <summary>
    /// Rechecks live account or agent state after bearer token validation.
    /// </summary>
    public override async Task TokenValidated(TokenValidatedContext context)
    {
        ClaimsPrincipal? principal = context.Principal;
        if (principal?.FindFirst("identity_type")?.Value != "vendor_user"
            || !Guid.TryParse(principal.FindFirst("vendor_id")?.Value, out Guid
vendorId)
            || !Guid.TryParse(principal.FindFirst("sub")?.Value, out Guid userId)
            || !Guid.TryParse(principal.FindFirst("sid")?.Value, out Guid sessionId))
        { context.Fail("Invalid tenant session."); return; }
        await using TenantContext? tenant = await authentication.ResolveAsync(vendorId,
context.HttpContext.RequestAborted);
        TenantAccess? access = tenant is null ? null : await
authentication.ValidateAsync(tenant, userId, sessionId, context.HttpContext.RequestAborted);
        if (access is null) { context.Fail("Invalid tenant session."); return; }
        // Never trust stale embedded role or permission claims: load current grants on
each request.
        ClaimsIdentity identity = (ClaimsIdentity)principal.Identity!;
        foreach (Claim? claim in
identity.FindAll("role").Concat(identity.FindAll("permission")).ToArray()) {
            identity.RemoveClaim(claim); }
        identity.AddClaims(access.Roles.Select(role => new Claim("role", role)));
        identity.AddClaims(access.Permissions.Select(permission => new Claim("permission",
permission)));
    }
}

```

```

    }
}

```

TenantRefreshCookie.cs

```

namespace BridgeControl.Security;

/// <summary>
/// Writes and clears the secure tenant refresh-credential cookie.
/// </summary>
public static class TenantRefreshCookie
{
    public const string Name = "__Secure-BridgeControl.Tenant.Refresh";
    public const string RequestHeader = "X-BridgeControl-Refresh";

    /// <summary>
    /// Creates the security and path settings for the refresh cookie.
    /// </summary>
    private static CookieOptions Options() => new() { HttpOnly = true, Secure = true, SameSite
= SameSiteMode.Strict, Path = "/api/v1/auth", IsEssential = true };

    /// <summary>
    /// Writes the refresh credential in a secure, HTTP-only cookie with its expiry.
    /// </summary>
    public static void Write(HttpResponse response, string credential, DateTimeOffset expiry)
    { CookieOptions options = Options(); options.Expires = expiry;
response.Cookies.Append(Name, credential, options); }

    /// <summary>
    /// Deletes the refresh credential cookie using its original path and security settings.
    /// </summary>
    public static void Clear(HttpResponse response) => response.Cookies.Delete(Name,
Options());
}

```

TenantServiceRegistration.cs

```

using System.Threading.RateLimiting;
using BridgeControl.Options;
using BridgeControl.Services.AgentServices;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Security;

/// <summary>
/// Registers tenant authentication and account services.
/// </summary>
public static class TenantServiceRegistration
{
    /// <summary>
    /// Registers tenant bearer authentication, authorization, and related services.
    /// </summary>
}

```

```

        public static IServiceCollection AddTenantAuthentication(this IServiceCollection services,
        IConfiguration configuration)
        {

services.AddOptions<TenantAuthenticationOptions>().Bind(configuration.GetSection(TenantAuthentica
tionOptions.SectionName))
            .ValidateDataAnnotations().Validate(x => x.HasValidSigningKey(), "Configure
Authentication: Tenant: SigningKey with a Base64 random key of at least 32 bytes.")
            .Validate<IOptions<PlatformAuthenticationOptions>>((tenant, platform) =>
tenant.SigningKey != platform.Value.SigningKey
            && tenant.Audience != platform.Value.Audience, "Tenant signing key
and audience must differ from platform authentication.").ValidateOnStart();
            services.AddScoped<TenantAuthenticationService>();

services.AddOptions<OperationalHistoryOptions>().Bind(configuration.GetSection("OperationalHistor
y")).ValidateDataAnnotations().ValidateOnStart();

services.AddScoped<BridgeControl.Repositories.Vendor.IOperationalHistoryRepository,
BridgeControl.Repositories.Vendor.OperationalHistoryRepository>();
            services.AddScoped<HistoryRetentionService>();
            services.AddHostedService<HistoryRetentionWorker>();
            services.AddScoped<AuditHistoryOperator>();
            services.AddScoped<TenantOperationFactory>();
            services.AddSingleton<LicenseEligibility>();
            services.AddScoped<BridgeControl.Repositories.Vendor.IInstallationRepository,
BridgeControl.Repositories.Vendor.InstallationRepository>();
            services.AddScoped<BridgeControl.Repositories.Vendor.ILicenseRepository,
BridgeControl.Repositories.Vendor.LicenseRepository>();

services.AddOptions<ArtifactArchiveOptions>().Bind(configuration.GetSection("ArtifactArchive")).V
alidateDataAnnotations().ValidateOnStart();
            services.AddSingleton<VendorArtifactArchive>();
            services.AddSingleton<ISchemaArtifactValidator,
Draft202012SchemaArtifactValidator>();
            services.AddSingleton<SchemaArtifactValidationService>();
            services.AddScoped<BridgeControl.Repositories.Vendor.ProductRepository>();
            services.AddScoped<DeploymentSettingsService>();
            services.AddScoped<AgentAssignmentService>();
            services.AddScoped<BridgeControl.Repositories.Vendor.ProductArtifactRepository>();
            services.AddSingleton<ConfigurationValues>();
            services.AddSingleton<ConfigurationArtifactReader>();

services.AddOptions<EffectiveConfigurationOptions>().Bind(configuration.GetSection("EffectiveConf
iguration")).ValidateDataAnnotations().ValidateOnStart();
            services.AddScoped<EffectiveConfigurationGenerator>();

services.AddOptions<StateReconciliationOptions>().Bind(configuration.GetSection("StateReconciliat
ion")).ValidateDataAnnotations().ValidateOnStart();
            services.AddScoped<InstallationStatePolicy>();
            services.AddScoped<StateReconciliationProcessor>();
            services.AddHostedService<StateReconciliationWorker>();
            services.AddScoped<BridgeControl.Repositories.Vendor.IInstallationStateRepository,
BridgeControl.Repositories.Vendor.InstallationStateRepository>();
            services.AddScoped<BridgeControl.Repositories.Vendor.IDeploymentRepository,
BridgeControl.Repositories.Vendor.DeploymentRepository>();
            services.AddScoped<ConfigurationMigrationRules>();
            services.AddScoped<ConfigurationMigrationPlanner>();
            services.AddScoped<ConfigurationMigrationService>();

services.AddScoped<BridgeControl.Repositories.Vendor.IConfigurationMigrationExecutionRepository,
BridgeControl.Repositories.Vendor.ConfigurationMigrationExecutionRepository>();

```

```

services.AddScoped<BridgeControl.Repositories.Vendor.IConfigurationMigrationRepository,
BridgeControl.Repositories.Vendor.ConfigurationMigrationRepository>();
services.AddScoped<EffectiveConfigurationProcessor>();
services.AddHostedService<EffectiveConfigurationWorker>();

services.AddScoped<BridgeControl.Repositories.Vendor.IEffectiveConfigurationRepository,
BridgeControl.Repositories.Vendor.EffectiveConfigurationRepository>();
services.AddScoped<BridgeControl.Repositories.Vendor.IConfigurationRepository,
BridgeControl.Repositories.Vendor.ConfigurationRepository>();
services.AddScoped<BridgeControl.Repositories.Vendor.IVersionPublishingRepository,
BridgeControl.Repositories.Vendor.VersionPublishingRepository>();
services.AddSingleton<RoleGrantPolicy>();
services.AddScoped<AccountInvitationService>();

services.AddScoped<BridgeControl.Repositories.Vendor.IUserAdministrationRepository,
BridgeControl.Repositories.Vendor.UserAdministrationRepository>();
services.AddScoped<BridgeControl.Repositories.Vendor.ICustomerRepository,
BridgeControl.Repositories.Vendor.CustomerRepository>();
services.AddScoped<BridgeControl.Repositories.Vendor.IAccessAssignmentRepository,
BridgeControl.Repositories.Vendor.AccessAssignmentRepository>();
services.AddScoped<TenantBearerEvents>();
services.AddSingleton<VendorSecurityAudit>();
services.AddScoped<TenantPasswordService>();
services.AddScoped<UserLifecycleService>();
services.AddScoped<VendorLifecycleService>();
services.AddSingleton<IPasswordResetDelivery, LocalPasswordResetDelivery>();
services.AddAuthentication().AddJwtBearer(TenantAuthentication.Scheme, _ => { });

services.AddOptions<JwtBearerOptions>(TenantAuthentication.Scheme).Configure<IOptions<TenantAuthen
ticationOptions>>((bearer, configured) =>
{
    TenantAuthenticationOptions options = configured.Value;
    bearer.MapInboundClaims = false; bearer.IncludeErrorDetails = false;
bearer.SaveToken = false;
    bearer.EventsType = typeof(TenantBearerEvents);
    bearer.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidIssuer = options.Issuer,
        ValidateAudience = true,
        ValidAudience = options.Audience,
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new
SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
        RequireSignedTokens = true,
        RequireExpirationTime = true,
        ValidateLifetime = true,
        ClockSkew = TimeSpan.Zero,
        ValidAlgorithms = [SecurityAlgorithms.HmacSha256],
        ValidTypes = ["at+jwt"],
        NameClaimType = "sub",
        RoleClaimType = "role"
    };
});
services.AddAuthorization(options =>
options.AddPolicy(TenantAuthentication.Policy, policy => policy

.AddAuthenticationSchemes(TenantAuthentication.Scheme).RequireAuthenticatedUser().RequireClaim("i
dentity_type", "vendor_user"));
services.AddRateLimiter(options =>
{

```

```

        foreach (string operation in new[] { "VendorDiscovery", "TenantLogin",
"TenantRefresh", "PasswordForgot", "PasswordReset", "PasswordChange" })
        {
            options.AddPolicy(operation, context =>
RateLimitPartition.GetFixedWindowLimiter($"{operation}:{context.Connection.RemoteIpAddress}", _
=>
                {
                    TenantAuthenticationOptions limits =
context.RequestServices.GetRequiredService<IOptions<TenantAuthenticationOptions>>().Value;
                    return new FixedWindowRateLimiterOptions
                    {
                        PermitLimit = operation switch
                        {
                            "VendorDiscovery" => 60, "TenantLogin" =>
limits.LoginRequestsPerMinute, "TenantRefresh" => limits.RefreshRequestsPerMinute,
"PasswordForgot" => 5, _ => 10 },

                        Window = TimeSpan.FromMinutes(1),
                        QueueLimit = 0,
                        AutoReplenishment = true
                    };
                });
        }
    });
    return services;
}
}

```

Services/AgentServices

AgentAssignmentService.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.AgentDTOS.DeploymentSettings;
using BridgeControl.DTOS.Shared;
using BridgeControl.DTOS.Vendor.DeploymentSettings;
using BridgeControl.Models.Vendor;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.AgentServices;

/// <summary>Builds application assignments from an agent's active installations and published
versions.</summary>
public sealed class AgentAssignmentService(ITenantResolver resolver, AgentIdentityService
identity)
{
    /// <summary>Lists application assignments available to the authenticated agent.</summary>
    public async Task<PagedResponse<AgentInstallationAssignment>> ListAsync(AgentActor actor,
PaginationQuery page, CancellationToken ct)
    {
        await using TenantContext tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct)
        ?? throw new ApiAccessException(401);
        Models.AgentModels.Agent agent = await identity.CurrentAsync(tenant, actor, ct)
    }
}

```

```

        ?? throw new ApiAccessException(401);

        IQueryable<Installation> query = tenant.Database.Installations.Where(installation
=>
            installation.AgentId == agent.Id
            && installation.CustomerId == agent.CustomerId
            && installation.Status == "Active"
            && installation.Product.Status == "Active");

        PagedResponse<Installation> installations = await query.ToPageAsync(page,
installation => installation.Id, ct);
        return await installations.MapAsync(
            (installation, token) => BuildAssignmentAsync(tenant.Database,
installation, token), ct);
    }

    private static async Task<AgentInstallationAssignment> BuildAssignmentAsync(
        VendorDbContext database, Installation installation, CancellationToken ct)
    {
        AgentApplication? application = await BuildApplicationAsync(database,
installation, ct);
        return new AgentInstallationAssignment(
            installation.Id, installation.CustomerId, installation.ProductId,
installation.AgentId,
            installation.Name, installation.Status, installation.RegisteredAt,
installation.LastSeenAt, application);
    }

    private static async Task<AgentApplication?> BuildApplicationAsync(
        VendorDbContext database, Installation installation, CancellationToken ct)
    {
        List<ProductVersion> publishedVersions = await database.ProductVersions
            .Where(version => version.ProductId == installation.ProductId
                && version.Status == "Published" && version.DeploymentSettingsJson
!= null)
            .OrderByDescending(version => version.PublishedAt)
            .ThenBy(version => version.Id)
            .ToListAsync(ct);

        Guid? desiredVersionId = await database.DesiredStateRevisions
            .Where(revision => revision.Id == installation.DesiredStateRevisionId)
            .Select(revision => (Guid?) revision.ProductVersionId)
            .SingleOrDefaultAsync(ct);

        // A desired version must be available; only installations without one use the
latest publication.
        ProductVersion? selectedVersion = desiredVersionId is null
            ? publishedVersions.FirstOrDefault()
            : publishedVersions.SingleOrDefault(version => version.Id ==
desiredVersionId);
        if (selectedVersion is null)
        {
            return null;
        }

        DeploymentSettings settings =
DeploymentSettingsService.Decode<DeploymentSettings>(selectedVersion.DeploymentSettingsJson)!;
        Guid?[] artifactIds = publishedVersions.Select(version =>
version.PublishedInstallerArtifactId).ToArray();
        List<FileArtifact> artifacts = await database.FileArtifacts
            .Where(artifact => artifactIds.Contains(artifact.Id))
            .ToListAsync(ct);
    }

```

```

        InstallerIdentity? selectedInstaller = ReadInstallerIdentity(selectedVersion,
artifacts);
        if (selectedInstaller?.UpgradeCode is not Guid upgradeCode || upgradeCode ==
Guid.Empty)
        {
            return null;
        }

        List<AgentVersionBinding> bindings = BuildVersionBindings(publishedVersions,
artifacts, upgradeCode);
        return new AgentApplication(
            installation.ProductId, selectedVersion.Id, installation.RegistrationName
?? installation.Name,
            settings.Kind, upgradeCode, settings.ConfigurationPath,
            settings.ExecutablePath ?? "", settings.ServiceName ?? "", bindings);
    }

    /// <summary>Includes only versions whose installers belong to the selected upgrade
family.</summary>
    private static List<AgentVersionBinding> BuildVersionBindings(
IEnumerable<ProductVersion> versions, List<FileArtifact> artifacts, Guid
upgradeCode)
    {
        List<AgentVersionBinding> bindings = [];
        foreach (ProductVersion version in versions)
        {
            InstallerIdentity? installer = ReadInstallerIdentity(version, artifacts);
            if (installer?.UpgradeCode == upgradeCode)
            {
                bindings.Add(new(version.Id, installer.ProductCode,
installer.Version));
            }
        }
        return bindings;
    }

    private static InstallerIdentity? ReadInstallerIdentity(ProductVersion version,
List<FileArtifact> artifacts)
    {
        FileArtifact? artifact = artifacts.SingleOrDefault(candidate => candidate.Id ==
version.PublishedInstallerArtifactId);
        return
DeploymentSettingsService.Decode<InstallerIdentity>(artifact?.InstallerIdentityJson);
    }
}

```

AgentCredentialOperator.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.Models.AgentModels;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.AgentServices;

```



```

/// <summary>Host-operator maintenance only; deliberately has no public HTTP
controller.</summary>
public sealed class AgentCredentialOperator(PlatformDbContext platform, ITenantResolver resolver,
    AgentIdentityService identity, TimeProvider clock)
{
    /// <summary>
    /// Rotates or revokes an agent machine credential and records the operation.
    /// </summary>
    public async Task<AgentRegistrationResponse?> ChangeAsync(Guid vendor, Guid agentId, bool
rotate, CancellationToken ct = default)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await resolver.ResolveForAgentAsync(vendor, ct)
?? throw new ApiAccessException(404);
        VendorDbContext db = tenant.Database;
        Agent? agent = await db.Agents.Include(x => x.Customer).SingleOrDefaultAsync(x =>
x.Id == agentId, ct);
        if (agent is null) { throw new ApiAccessException(404); }
        if (rotate && (agent.Status != "Active" || agent.Customer.Status != "Active"))
        { throw new ApiConflictException("inactive-agent", "Only an active Agent and
customer can receive a new credential."); }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        DateTimeOffset now = clock.GetUtcNow();
        await db.AgentCredentials.Where(x => x.AgentId == agentId && x.RevokedAt ==
null).ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, now), ct);
        AgentRegistrationResponse? issued = rotate ? identity.Issue(db, vendor, agentId) :
null;

        db.VendorAuditLogs.Add(new VendorAuditLog
        {
            ActorType = "Operator",
            ActorId = Guid.Empty,
            Action = rotate ? "AgentCredentialRotated" : "AgentCredentialRevoked",
            EntityType = "Agent",
            EntityId = agentId,
            Outcome = "Succeeded",
            CreatedAt = now,
            Context = JsonSerializer.Serialize(new { mechanism = "offline-operator",
operatorAccount = Environment.UserName, operatorHost = Environment.MachineName })
        });
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return issued;
    }
    /// <summary>
    /// Prompts for an agent credential operation and displays its result.
    /// </summary>
    public async Task<int> RunInteractiveAsync(CancellationToken ct = default)
    {
        if (Console.IsInputRedirected || Console.IsOutputRedirected)
        { Console.Error.WriteLine("Use a private interactive terminal; credential output
must not be redirected."); return 1; }
        Console.WriteLine("Stop all API instances before Agent credential maintenance. The
API server will not start.");
        Console.Write("Type ROTATE or REVOKE to confirm the API is stopped and choose the
operation: ");
        string? action = Console.ReadLine(); if (action is not ("ROTATE" or "REVOKE")) {
return 1; }
        Console.Write("Vendor ID: "); if (!Guid.TryParse(Console.ReadLine(), out Guid
vendor)) { return 1; }
        Console.Write("Agent ID: "); if (!Guid.TryParse(Console.ReadLine(), out Guid
agent)) { return 1; }
    }
}

```

```

        AgentRegistrationResponse? issued = await ChangeAsync(vendor, agent, action ==
"ROTATE", ct);
        Console.WriteLine("Existing credentials and their tokens are revoked. Identity and
installation ownership are preserved.");
        if (issued is not null)
        {
            Console.WriteLine("Store this replacement credential privately on the
existing Agent. It is displayed once:");
            Console.WriteLine(issued.Credential); Console.WriteLine($"Expires:
{issued.ExpiresAt:O}");
        }
        return 0;
    }
}

```

AgentIdentityService.cs

```

using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Security.Cryptography;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.AgentDTOS.AgentContracts;
using BridgeControl.Models.AgentModels;
using BridgeControl.Options;
using BridgeControl.Security.AgentSecurity;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Services.AgentServices;
/// <summary>
/// Coordinates agent identity operations.
/// </summary>
public sealed class AgentIdentityService(PlatformDbContext platform, ITenantResolver resolver,
    TenantOperationFactory operations, RefreshCredentialService secrets, AgentRateLimits
limits,
    VendorSecurityAudit audit, IOptions<AgentAuthenticationOptions> configuration,
    TimeProvider clock)
{
    /// <summary>
    /// Creates a bounded, single-use enrollment credential for an authorized agent
    registration.
    /// </summary>
    public async Task<AgentEnrollmentResponse> EnrollAsync(TenantActor actor,
CreateAgentEnrollmentRequest request, Cancellation token ct)
    {
        limits.Check("enroll", actor.VendorId, actor.UserId);
        await using TenantOperation operation = await operations.OpenAsync(actor,
"agents.manage", true, ct);
        VendorDbContext db = operation.Database;
        if (!await operation.Access.Customers(db, "agents.manage").AnyAsync(x => x.Id ==
request.CustomerId && x.Status == "Active", ct))
        { throw new ApiAccessException(404); }
        string secret = secrets.Create(); secrets.TryHash(secret, out byte[] hash);
        AgentEnrollment enrollment = new()

```

```

        {
            CustomerId = request.CustomerId,
            CreatedBy = actor.UserId,
            SecretHash = hash,
            CreatedAt = clock.GetUtcNow(),
            ExpiresAt = clock.GetUtcNow().AddMinutes(15)
        };
        db.AgentEnrollments.Add(enrollment);
        audit.Record(db, actor.UserId, "AgentEnrollmentCreated", enrollment.Id,
entityType: "AgentEnrollment");
        await db.SaveChangesAsync(ct);
        return new(Encode("bce1", actor.VendorId, enrollment.Id, secret),
enrollment.ExpiresAt);
    }

    /// <summary>
    /// Consumes an enrollment credential and creates the agent's persistent machine identity.
    /// </summary>
    public async Task<AgentRegistrationResponse> RegisterAsync(RegisterAgentRequest request,
CancellationToken ct)
    {
        if (!TryParse(request.EnrollmentCode, "bce1", out Guid vendor, out Guid id, out
byte[] hash)) { throw new ApiAccessException(401); }
        limits.Check("register", vendor, id);
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await resolver.ResolveForAgentAsync(vendor, ct)
?? throw new ApiAccessException(401);
        VendorDbContext db = tenant.Database;
        AgentEnrollment? enrollment = await db.AgentEnrollments.Include(x =>
x.Customer).SingleOrDefaultAsync(x => x.Id == id, ct);
        DateTimeOffset now = clock.GetUtcNow();
        DateTimeOffset? cutoff = await CutoffAsync(vendor, ct);
        bool matches = CryptographicOperations.FixedTimeEquals(hash,
enrollment?.SecretHash ?? new byte[32]);
        if (!matches || enrollment is null || enrollment.Purpose != "AgentEnrollment" ||
enrollment.UsedAt is not null
            || enrollment.RevokedAt is not null || enrollment.ExpiresAt <= now ||
enrollment.Customer.Status != "Active"
            || (cutoff is not null && enrollment.CreatedAt <= cutoff))
        {
            audit.Record(db, Guid.Empty, "AgentRegistrationDenied", id, "Denied",
"AgentEnrollment", "Agent");
            await db.SaveChangesAsync(ct); throw new ApiAccessException(401);
        }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        Agent agent = new() { CustomerId = enrollment.CustomerId, MachineName =
request.MachineName, AgentVersion = request.AgentVersion, RegisteredAt = now };
        db.Agents.Add(agent);
        AgentRegistrationResponse issued = Issue(db, vendor, agent.Id);
        enrollment.UsedAt = now; enrollment.AgentId = agent.Id;
        audit.Record(db, agent.Id, "AgentRegistered", agent.Id, entityType: "Agent",
actorType: "Agent");
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
        return issued;
    }

    /// <summary>
    /// Exchanges a valid machine credential for a short-lived agent access token.
    /// </summary>

```

```

        public async Task<AgentTokenResponse> TokenAsync (AgentTokenRequest request,
CancellationToken ct)
        {
            if (!TryParse(request.Credential, "bcal", out Guid vendor, out Guid id, out byte[]
hash)) { throw new ApiAccessException(401); }
            limits.Check("token", vendor, id);
            await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
            await using TenantContext tenant = await resolver.ResolveForAgentAsync(vendor, ct)
?? throw new ApiAccessException(401);
            AgentCredential? credential = await
tenant.Database.AgentCredentials.AsNoTracking().SingleOrDefaultAsync(x => x.Id == id, ct);
            bool matches = CryptographicOperations.FixedTimeEquals(hash,
credential?.SecretHash ?? new byte[32]);
            if (!matches || credential is null || await CurrentAsync(tenant, new(vendor,
credential.AgentId, id), ct) is null)
            {
                audit.Record(tenant.Database, Guid.Empty, "AgentAuthenticationDenied", id,
"Denied", "AgentCredential", "Agent");
                await tenant.Database.SaveChangesAsync(ct); throw new
ApiAccessException(401);
            }
            DateTimeOffset now = clock.GetUtcNow();
            DateTimeOffset expires = now.AddMinutes(10) < credential.ExpiresAt ?
now.AddMinutes(10) : credential.ExpiresAt;
            AgentAuthenticationOptions options = configuration.Value;
            JwtSecurityToken token = new(options.Issuer, options.Audience,
[new Claim("sub", credential.AgentId.ToString()), new Claim("vendor_id",
vendor.ToString()),
new Claim("credential_id", id.ToString()), new Claim("identity_type",
"agent"), new Claim("jti", Guid.NewGuid().ToString("N"))],
now.UtcDateTime, expires.UtcDateTime, new SigningCredentials(new
SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
SecurityAlgorithms.HmacSha256));
            token.Header["typ"] = "at+jwt";
            audit.Record(tenant.Database, credential.AgentId, "AgentTokenIssued", id,
entityType: "AgentCredential", actorType: "Agent");
            await tenant.Database.SaveChangesAsync(ct);
            return new(new JwtSecurityTokenHandler().WriteToken(token), expires);
        }

        /// <summary>
        /// Rechecks an agent identity against live lifecycle and credential state.
        /// </summary>
        public async Task<bool> ValidateAsync (AgentActor actor, CancellationToken ct)
        {
            await using TenantContext? tenant = await
resolver.ResolveForAgentAsync(actor.VendorId, ct);
            return tenant is not null && await CurrentAsync(tenant, actor, ct) is not null;
        }

        /// <summary>
        /// Loads the current agent credential subject to expiry and revocation checks.
        /// </summary>
        public async Task<Agent?> CurrentAsync (TenantContext tenant, AgentActor actor,
CancellationToken ct)
        {
            if (tenant.VendorId != actor.VendorId) { return null; }
            DateTimeOffset now = clock.GetUtcNow(); DateTimeOffset? cutoff = await
CutoffAsync(actor.VendorId, ct);
            return await tenant.Database.AgentCredentials.Where(x => x.Id ==
actor.CredentialId && x.AgentId == actor.AgentId

```

```

        && x.Purpose == "AgentAuthentication" && x.RevokedAt == null && x.ExpiresAt
> now
        && (cutoff == null || x.CreatedAt > cutoff) && x.Agent.Status == "Active"
&& x.Agent.Customer.Status == "Active")
        .Select(x => x.Agent).SingleOrDefaultAsync(ct);
    }

    // Caller owns authorization, a vendor write lock, transaction and audit.
    /// <summary>
    /// Creates and records a new agent machine credential and its public response.
    /// </summary>
    internal AgentRegistrationResponse Issue(VendorDbContext db, Guid vendor, Guid agent)
    {
        string secret = secrets.Create(); secrets.TryHash(secret, out byte[] hash);
        AgentCredential credential = new() { AgentId = agent, SecretHash = hash, CreatedAt
= clock.UtcNow(), ExpiresAt = clock.UtcNow().AddDays(90) };
        db.AgentCredentials.Add(credential);
        return new(agent, Encode("bca1", vendor, credential.Id, secret),
credential.ExpiresAt);
    }
    /// <summary>
    /// Reads the vendor's credential revocation cutoff.
    /// </summary>
    private Task<DateTimeOffset?> CutoffAsync(Guid vendor, CancellationToken ct) =>
platform.SoftwareVendors.AsNoTracking()
        .Where(x => x.Id == vendor).Select(x => x.SessionsRevokedBefore).SingleAsync(ct);
    /// <summary>
    /// Formats a purpose-bound credential with vendor and credential identifiers.
    /// </summary>
    private static string Encode(string purpose, Guid vendor, Guid id, string secret) =>
$"{purpose}.{vendor:N}.{id:N}.{secret}";
    /// <summary>
    /// Parses a credential of the expected purpose and hashes its secret component.
    /// </summary>
    private bool TryParse(string value, string purpose, out Guid vendor, out Guid id, out
byte[] hash)
    {
        vendor = id = Guid.Empty; hash = [];
        string[] parts = value.Split('.');
        return parts.Length == 4 && parts[0] == purpose && Guid.TryParseExact(parts[1],
"N", out vendor) && vendor != Guid.Empty
        && Guid.TryParseExact(parts[2], "N", out id) && id != Guid.Empty &&
secrets.TryHash(parts[3], out hash);
    }
}

```

AgentRateLimits.cs

```

using System.Threading.RateLimiting;
using BridgeControl.Api.Exceptions;

namespace BridgeControl.Services.AgentServices;

/// <summary>Per-account/credential limits complement the endpoint IP limit, including
failures.</summary>
public sealed class AgentRateLimits : IDisposable
{
    private readonly PartitionedRateLimiter<string> limiter =
PartitionedRateLimiter.Create<string, string>(key =>

```

```

        RateLimitPartition.GetFixedWindowLimiter(key, _ => new
FixedWindowRateLimiterOptions
    { PermitLimit = key.StartsWith("token:", StringComparison.Ordinal) ? 60 : 10,
Window = TimeSpan.FromMinutes(1), QueueLimit = 0 }));
    /// <summary>
    /// Enforces the request budget for the selected agent identity operation.
    /// </summary>
    public void Check(string operation, Guid vendor, Guid id)
    {
        using RateLimitLease lease =
limiter.AttemptAcquire($"{operation}:{vendor:N}:{id:N}");
        if (!lease.IsAcquired) { throw new AgentRateLimitException(); }
    }
    /// <summary>
    /// Releases the resources owned by this instance.
    /// </summary>
    public void Dispose() => limiter.Dispose();
}

```

Services/Platform

DatabaseHostLease.cs

```

using BridgeControl.Data;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Platform;

/// <summary>One database-scoped session lock excludes API hosts and offline operators from each
other.</summary>
public sealed class DatabaseHostLease : IAsyncDisposable
{
    private readonly SqlConnection connection;

    /// <summary>
    /// Initializes a new <see cref="DatabaseHostLease"/> instance and its dependencies.
    /// </summary>
    private DatabaseHostLease(SqlConnection connection) => this.connection = connection;

    /// <summary>
    /// Acquires the database host lease that excludes offline maintenance.
    /// </summary>
    public static async Task<DatabaseHostLease> AcquireAsync(PlatformDbContext platform,
CancellationTokens ct)
    {
        SqlConnection connection = new(new
SqlConnectionStringBuilder(platform.Database.GetConnectionString()) { Pooling = false,
ConnectRetryCount = 0 }.ConnectionString);
        try
        {
            await connection.OpenAsync(ct);
            using SqlCommand command = connection.CreateCommand();
            command.CommandText = "DECLARE @result int; EXEC @result =
sys.sp_getapplock @Resource=N'BridgeControl:HostOrOperator', @LockMode='Exclusive',
@LockOwner='Session', @LockTimeout=0; SELECT @result;";

```

```

        if ((int)(await command.ExecuteScalarAsync(ct))! < 0) { throw new
InvalidOperationException("Stop the API and other operator processes before maintenance. Only one
API host is supported."); }
        return new(connection);
    }
    catch { await connection.DisposeAsync(); throw; }
}

/// <summary>
/// Checks that the connection holding the host lease remains usable.
/// </summary>
public async Task CheckAsync(CancellationToken ct)
{
    using SqlCommand command = connection.CreateCommand(); command.CommandTimeout = 5;
    command.CommandText = "SELECT APPLOCK_MODE('public',
'BridgeControl:HostOrOperator', 'Session');";
    if (!string.Equals(await command.ExecuteScalarAsync(ct) as string, "Exclusive",
StringComparison.Ordinal)) { throw new InvalidOperationException("Host lease lost."); }
}

/// <summary>
/// Asynchronously releases the resources owned by this instance.
/// </summary>
public ValueTask DisposeAsync() => connection.DisposeAsync();
}

```

DatabaseHostLeaseService.cs

```

using BridgeControl.Data;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Coordinates database host lease operations.
/// </summary>
public sealed class DatabaseHostLeaseService(IServiceScopeFactory scopes,
IHostApplicationLifetime lifetime) : IHostedService, IDisposable
{
    private DatabaseHostLease? lease;
    private readonly CancellationTokenSource stopping = new();
    private Task? monitor;

    /// <summary>
    /// Acquires the host lease and starts lease monitoring.
    /// </summary>
    public async Task StartAsync(CancellationToken ct)
    {
        await using AsyncServiceScope scope = scopes.CreateAsyncScope();
        lease = await
DatabaseHostLease.AcquireAsync(scope.ServiceProvider.GetRequiredService<PlatformDbContext>(),
ct);

        monitor = MonitorAsync();
    }

    /// <summary>
    /// Checks the host lease periodically and stops the host if it is lost.
    /// </summary>
    private async Task MonitorAsync()
    {
        try

```

```

        {
            while (!stopping.IsCancellationRequested)
            { await Task.Delay(TimeSpan.FromSeconds(2), stopping.Token); await
lease!.CheckAsync(stopping.Token); }
        }
        catch (OperationCanceledException) when (stopping.IsCancellationRequested) { }
        catch { lifetime.StopApplication(); }
    }

    /// <summary>
    /// Stops lease monitoring and releases the host lease.
    /// </summary>
    public async Task StopAsync(CancellationToken ct)
    {
        await stopping.CancelAsync();
        if (monitor is not null) { await monitor; }
        if (lease is not null) { await lease.DisposeAsync(); }
    }

    /// <summary>
    /// Releases the resources owned by this instance.
    /// </summary>
    public void Dispose() => stopping.Dispose();
}

```

DatabaseOperationsOperator.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Platform;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Platform;

/// <summary>Local OS-authorized tooling; never exposed through HTTP.</summary>
public sealed class DatabaseOperationsOperator(VendorDatabaseOperations operations,
PlatformDbContext platform, IConfiguration configuration)
{
    /// <summary>
    /// Executes the requested offline database maintenance command.
    /// </summary>
    public async Task<int> RunAsync(bool scheduled, CancellationToken ct = default)
    {
        if (scheduled)
        {
            if
(!Guid.TryParse(configuration["DatabaseOperations:ScheduledAdministratorId"], out Guid
administrator))
            { throw new ArgumentException("Configure the scheduled operation's active
platform administrator ID."); }
            int failures = await operations.BackupDueAsync(administrator, ct);
            await operations.ExpireBackupsAsync(administrator, ct);
            Console.WriteLine($"Daily backup pass completed; {failures} vendor(s) need
operator attention.");
            return failures == 0 ? 0 : 1;
        }
        Console.Write("Active platform administrator ID: "); Guid admin = ReadGuid();
        if (!await platform.PlatformAdmins.AnyAsync(x => x.Id == admin && x.Status ==
"Active", ct)) { throw new ArgumentException("An active platform administrator is required."); }
        Console.Write("Vendor ID: "); Guid vendor = ReadGuid();
    }
}

```



```

        Console.WriteLine("Operation (Upgrade / Backup / Restore / Status): "); string kind =
        Console.ReadLine()?.Trim() ?? "";
        if (kind == "Status")
        {
            foreach (VendorMaintenanceOperation? operation in await
platform.VendorMaintenanceOperations.AsNoTracking().Where(x => x.VendorId ==
vendor).OrderByDescending(x => x.CreatedAt).Take(50).ToArrayAsync(ct))
            { Console.WriteLine($"{operation.Id} {operation.Kind} {operation.Status}
{operation.Stage} schema={operation.CurrentSchema} target={operation.TargetSchema}
failure={operation.FailureCode ?? "none"}"); }
            return 0;
        }
        Console.WriteLine("Operation ID to resume, or blank for a new operation: "); string
input = Console.ReadLine()?.Trim() ?? "";
        Guid id = input.Length == 0 ? Guid.NewGuid() : Guid.Parse(input);
        Guid? backup = null;
        if (kind == "Restore")
        {
            Console.WriteLine("Completed backup operation ID: "); backup = ReadGuid();
            Console.WriteLine("Restore replaces this vendor's database and referenced
archive contents, and revokes existing sessions/Agent credentials.");
            Console.WriteLine($"Type RESTORE {vendor} to confirm: ");
            if (Console.ReadLine() != $"RESTORE {vendor}") { throw new
ArgumentException("Restore confirmation did not match."); }
        }
        Console.WriteLine($"Operation ID: {id}. Keep this ID to resume after
interruption.");
        VendorMaintenanceOperation result = await operations.ExecuteAsync(admin, vendor,
id, kind, backup, ct);
        Console.WriteLine($"{result.Id} {result.Status}; vendor schema
{result.CurrentSchema}."); return 0;
    }

    /// <summary>
    /// Reads and validates an identifier entered for a maintenance command.
    /// </summary>
    private static Guid ReadGuid() => Guid.TryParse(Console.ReadLine(), out Guid id) && id !=
Guid.Empty ? id : throw new ArgumentException("A nonempty identifier is required.");
}

```

EncryptedBackup.cs

```

using System.Security.Cryptography;
using System.Text;

namespace BridgeControl.Services.Platform;

/// <summary>Versioned encrypt-then-MAC envelope. Authentication completes before any plaintext
is released.</summary>
public static class EncryptedBackup
{
    private static readonly byte[] Magic = "BCBACK01"u8.ToArray();
    private const int HeaderLength = 88;

    /// <summary>
    /// Encrypts a backup stream using authenticated encryption.
    /// </summary>
    public static async Task EncryptAsync(string source, string destination, string keyId,
byte[] key, long limit, CancellationToken ct)
    {

```

```

        if (key.Length != 32 || keyId.Length is < 1 or > 64 || keyId.Any(c =>
!char.IsAsciiLetterOrDigit(c) && c != '-'))
        { throw new InvalidOperationException("Invalid backup key configuration."); }
        byte[] header = new byte[HeaderLength];
        Magic.CopyTo(header, 0);
        Encoding.ASCII.GetBytes(keyId).CopyTo(header, 8);
        RandomNumberGenerator.Fill(header.AsSpan(72, 16));
        using Aes aes = Aes.Create();
        aes.Key = Derive(key, "encryption"); aes.IV = header[72..];
        await using (FileStream output = new(destination, FileMode.CreateNew,
FileAccess.Write, FileShare.None))
        {
            await output.WriteAsync(header, ct);
            await using CryptoStream encrypted = new(output, aes.CreateEncryptor(),
CryptoStreamMode.Write, leaveOpen: true);
            await using FileStream input = File.OpenRead(source);
            if (input.Length > limit) { throw new InvalidOperationException("Backup
exceeds the configured limit."); }
            await input.CopyToAsync(encrypted, ct);
            await encrypted.FlushFinalBlockAsync(ct);
        }
        byte[] tag;
        using (HMACSHA256 mac = new(Derive(key, "authentication")))
        await using (FileStream input = File.OpenRead(destination)) { tag = await
mac.ComputeHashAsync(input, ct); }
        await using FileStream append = new(destination, FileMode.Append,
FileAccess.Write, FileShare.None);
        await append.WriteAsync(tag, ct); await append.FlushAsync(ct); append.Flush(true);
    }

    /// <summary>
    /// Authenticates and decrypts a backup stream.
    /// </summary>
    public static async Task DecryptAsync(string source, string destination,
IReadOnlyDictionary<string, string> keys, long limit, CancellationToken ct)
    {
        await using FileStream input = new(source, FileMode.Open, FileAccess.Read,
FileShare.Read);
        if (input.Length < HeaderLength + 48 || input.Length > limit + HeaderLength + 48)
        { throw new InvalidDataException("Invalid backup length."); }
        byte[] header = new byte[HeaderLength]; await input.ReadExactlyAsync(header, ct);
        if (!header.AsSpan(0, 8).SequenceEqual(Magic)) { throw new
InvalidDataException("Unsupported backup format."); }
        string id = Encoding.ASCII.GetString(header, 8, 64).TrimEnd('\0');
        if (!keys.TryGetValue(id, out string? secret)) { throw new
InvalidDataException("Backup decryption key unavailable."); }
        byte[] key = Convert.FromBase64String(secret);
        if (key.Length != 32) { throw new InvalidDataException("Invalid backup key."); }
        using IncrementalHash mac = IncrementalHash.CreateHMAC(HashAlgorithmName.SHA256,
Derive(key, "authentication"));
        input.Position = 0;
        byte[] buffer = new byte[81920]; long remaining = input.Length - 32;
        while (remaining > 0)
        {
            int read = await input.ReadAsync(buffer.AsMemory(0,
(int)Math.Min(buffer.Length, remaining)), ct);
            if (read == 0) { throw new EndOfStreamException(); }
            mac.AppendData(buffer, 0, read); remaining -= read;
        }
        byte[] expected = new byte[32]; await input.ReadExactlyAsync(expected, ct);
        if (!CryptographicOperations.FixedTimeEquals(mac.GetHashAndReset(), expected)) {
throw new InvalidDataException("Backup authentication failed."); }
    }

```

```

        input.Position = HeaderLength;
        using Aes aes = Aes.Create(); aes.Key = Derive(key, "encryption"); aes.IV =
header[72..];
        await using FileStream output = new(destination, FileMode.CreateNew,
FileAccess.Write, FileShare.None);
        await using CryptoStream plaintext = new(output, aes.CreateDecryptor(),
CryptoStreamMode.Write, leaveOpen: true);
        remaining = input.Length - HeaderLength - 32;
        while (remaining > 0)
        {
            int read = await input.ReadAsync(buffer.AsMemory(0,
(int)Math.Min(buffer.Length, remaining)), ct);
            if (read == 0) { throw new EndOfStreamException(); }
            await plaintext.WriteAsync(buffer.AsMemory(0, read), ct); remaining -=
read;
        }
        await plaintext.FlushFinalBlockAsync(ct);
        CryptographicOperations.ZeroMemory(key);
    }

    /// <summary>
    /// Derives the backup encryption key for the supplied context.
    /// </summary>
    private static byte[] Derive(byte[] key, string purpose) =>
HKDF.DeriveKey(HashAlgorithmName.SHA256, key, 32, info: Encoding.ASCII.GetBytes("BridgeControl
backup v1 " + purpose));
}

```

IVendorDatabaseMaintenance.cs

```

using BridgeControl.Data;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Defines the contract for vendor database maintenance operations.
/// </summary>
public interface IVendorDatabaseMaintenance
{
    /// <summary>
    /// Applies pending vendor database migrations.
    /// </summary>
    Task MigrateAsync(VendorDbContext database, string migration, CancellationToken ct);

    /// <summary>
    /// Creates a native SQL Server backup of the vendor database.
    /// </summary>
    Task BackupAsync(VendorDbContext database, string path, CancellationToken ct);

    /// <summary>
    /// Restores the vendor database from a native SQL Server backup.
    /// </summary>
    Task RestoreAsync(VendorDbContext database, string path, CancellationToken ct);

    /// <summary>
    /// Checks that the restored vendor database can be used.
    /// </summary>
    Task VerifyAsync(VendorDbContext database, CancellationToken ct);
}

```

IVendorInfrastructure.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Platform;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Opens only server-named vendor resources and prepares their private file archive.
/// </summary>
public interface IVendorInfrastructure
{
    /// <summary>
    /// Opens a context for the vendor database.
    /// </summary>
    VendorDbContext OpenDatabase(SoftwareVendor vendor);

    /// <summary>
    /// Resolves the vendor-specific artifact archive path.
    /// </summary>
    string GetArchivePath(SoftwareVendor vendor);

    /// <summary>
    /// Creates and protects the vendor artifact archive directory.
    /// </summary>
    void EnsureArchive(SoftwareVendor vendor);
}
```

PlatformAccessTokenService.cs

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using BridgeControl.DTOs.Platform;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Shared;
using BridgeControl.Options;
using BridgeControl.Security;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Issues signed access tokens that identify a platform administrator and their server-side session.
/// </summary>
public class PlatformAccessTokenService(IOptions<PlatformAuthenticationOptions> configuration,
    TimeProvider clock)
{
    /// <summary>
    /// Creates a signed platform access token for the administrator session.
    /// </summary>
    public PlatformAuthSessionResponse Create(PlatformAdmin administrator,
        AuthenticationSession session)
    {
        PlatformAuthenticationOptions options = configuration.Value;
        DateTimeOffset now = clock.GetUtcNow();
        DateTimeOffset expiry = now.AddMinutes(options.AccessTokenMinutes);
        if (expiry > session.ExpiresAt)
```

```

        {
            expiry = session.ExpiresAt;
        }
        expiry = DateTimeOffset.FromUnixTimeSeconds(expiry.ToUnixTimeSeconds());
        Claim[] claims =
        [
            new(JwtRegisteredClaimNames.Sub, administrator.Id.ToString()),
            new(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
            new(PlatformAuthentication.SessionIdClaim, session.Id.ToString()),
            new(PlatformAuthentication.IdentityTypeClaim,
PlatformAuthentication.IdentityType),
            new("role", PlatformAuthentication.Role)
        ];
        JwtSecurityToken token = new(options.Issuer, options.Audience, claims,
            now.UtcDateTime, expiry.UtcDateTime,
            new SigningCredentials(new
SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
                SecurityAlgorithms.HmacSha256));
        token.Header["typ"] = "at+jwt";
        return new PlatformAuthSessionResponse
        {
            PlatformAdminId = administrator.Id,
            Email = administrator.Email,
            AccessToken = new JwtSecurityTokenHandler().WriteToken(token),
            AccessTokenExpiresAt = expiry,
            Roles = [PlatformAuthentication.Role]
        };
    }
}

```

PlatformAdminRecovery.cs

```

using System.Data;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Resets an existing administrator's password and revokes their sessions through an offline
operator command.
/// </summary>
public sealed class PlatformAdminRecovery(PlatformDbContext database,
    PasswordService<PlatformAdmin> passwords, TimeProvider clock)
{
    // Run with API instances stopped: an offline operator owns database access for this
maintenance operation.
    /// <summary>
    /// Resets an existing administrator password and atomically revokes its sessions.
    /// </summary>
    public async Task<bool> ResetAsync(string email, string password, CancellationToken
cancellation_token = default)
    {
        email = AccountValidation.NormalizeEmail(email);
        string? error = AccountValidation.ValidateNewAccount(email, password);
        if (error is not null)
    }
}

```

```

        {
            throw new ArgumentException(error);
        }
        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(IsolationLevel.Serializable, cancellationToken);
        PlatformAdmin? administrator = await
database.PlatformAdmins.SingleOrDefaultAsync(admin => admin.Email == email, cancellationToken);
        if (administrator is null)
        {
            return false;
        }
        administrator.PasswordHash = passwords.Hash(administrator, password);
        DateTimeOffset now = clock.GetUtcNow();
        await database.AuthenticationSessions.Where(session => session.AccountId ==
administrator.Id && session.RevokedAt == null)
            .ExecuteUpdateAsync(setters => setters.SetProperty(session =>
session.RevokedAt, now), cancellationToken);
        database.PlatformAuditLogs.Add(new PlatformAuditLog
        {
            PlatformAdminId = administrator.Id,
            Action = "PlatformAdministratorPasswordRecovered",
            EntityType = nameof(PlatformAdmin),
            EntityId = administrator.Id.ToString(),
            Details = JsonSerializer.Serialize(new { mechanism = "offline-operator",
operatorAccount = Environment.UserName, operatorHost = Environment.MachineName }),
            CreatedAt = now
        });
        await database.SaveChangesAsync(cancellationToken);
        await transaction.CommitAsync(cancellationToken);
        return true;
    }

    /// <summary>
    /// Prompts for administrator recovery credentials and reports the reset outcome.
    /// </summary>
    public async Task<int> RunInteractiveAsync(CancellationTokencancellationTokencancellationToken = default)
    {
        if (Console.IsInputRedirected)
        {
            Console.Error.WriteLine("Run recovery in an interactive terminal so the
password can be entered privately.");
            return 1;
        }
        Console.WriteLine("Stop all BridgeControl API instances before offline password
recovery. This command does not start an API server.");
        Console.Write("Type RESET to confirm that the API is stopped and you intend to
reset an existing administrator: ");
        if (Console.ReadLine() != "RESET")
        {
            return 1;
        }
        Console.Write("Administrator email: ");
        string email = Console.ReadLine() ?? string.Empty;
        Console.Write($"New password
({AccountValidation.MinimumPasswordLength}-{AccountValidation.MaximumPasswordLength}
characters; input hidden): ");
        string password = InteractiveCredentials.ReadPassword();
        Console.Write("Confirm new password: ");
        if (password != InteractiveCredentials.ReadPassword())
        {
            Console.Error.WriteLine("The passwords do not match. No account was
changed.");

```

```

        return 1;
    }
    bool changed = await ResetAsync(email, password, cancellationTokens);
    Console.WriteLine(changed
        ? "Password changed and all sessions for this administrator revoked.
Account status was preserved."
        : "No matching administrator exists. No account was created or changed.");
    return changed ? 0 : 1;
}
}

```

PlatformAdminSetup.cs

```

using System.Data;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Security;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Creates the first platform administrator through a manually invoked local setup command.
/// </summary>
public class PlatformAdminSetup(PlatformDbContext database, PasswordService<PlatformAdmin>
passwords, TimeProvider clock)
{
    /// <summary>
    /// Creates the initial platform administrator only when none already exists.
    /// </summary>
    public async Task<bool> CreateAsync(string email, string password, CancellationToken
cancellationTokens = default)
    {
        email = AccountValidation.NormalizeEmail(email);
        string? error = AccountValidation.ValidateNewAccount(email, password);
        if (error is not null)
        {
            throw new ArgumentException(error);
        }
        PlatformAdmin administrator = new()
        {
            Id = Guid.NewGuid(),
            Email = email,
            Status = PlatformAuthentication.ActiveStatus,
            CreatedAt = clock.GetUtcNow()
        };
        administrator.PasswordHash = passwords.Hash(administrator, password);

        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(IsolationLevel.Serializable, cancellationTokens);
        // Serialize this operation even if two setup processes start against an empty
table.
        await database.Database.ExecuteSqlRawAsync("""
            DECLARE @result int;
            EXEC @result = sys.sp_getapplock
                @Resource = N'BridgeControl.Platform.FirstAdministrator',
                @LockMode = N'Exclusive', @LockOwner = N'Transaction', @LockTimeout
= 10000;

```

```

        IF @result < 0 THROW 51003, 'Another administrator setup is in progress.
Try again later.', 1;
        "", cancellationToken);
    if (await database.PlatformAdmins.AnyAsync(cancellationToken))
    {
        return false;
    }
    database.PlatformAdmins.Add(administrator);
    database.PlatformAuditLogs.Add(new PlatformAuditLog
    {
        PlatformAdminId = administrator.Id,
        Action = "FirstAdministratorCreated",
        EntityType = nameof(PlatformAdmin),
        EntityId = administrator.Id.ToString(),
        Details = "{}",
        CreatedAt = administrator.CreatedAt
    });
    await database.SaveChangesAsync(cancellationToken);
    await transaction.CommitAsync(cancellationToken);
    return true;
}

/// <summary>
/// Prompts for credentials and creates the initial platform administrator.
/// </summary>
public async Task<int> RunInteractiveAsync(CancellationTokentoken cancellationToken = default)
{
    if (await database.PlatformAdmins.AnyAsync(cancellationToken))
    {
        Console.WriteLine("Setup stopped: a platform administrator already exists.
No account was changed.");
        return 1;
    }
    if (Console.IsInputRedirected)
    {
        Console.Error.WriteLine("Run setup in an interactive terminal so the
password can be entered without echoing it.");
        return 1;
    }
    Console.Write("Administrator email: ");
    string email = Console.ReadLine() ?? string.Empty;
    Console.Write($"Password
({AccountValidation.MinimumPasswordLength}-{AccountValidation.MaximumPasswordLength} characters;
input hidden): ");
    string password = InteractiveCredentials.ReadPassword();
    Console.Write("Confirm password: ");
    string confirmation = InteractiveCredentials.ReadPassword();
    if (password != confirmation)
    {
        Console.Error.WriteLine("The passwords do not match. No account was
created.");
        return 1;
    }
    bool created = await CreateAsync(email, password, cancellationToken);
    Console.WriteLine(created
        ? "The first platform administrator was created. Start the API to log in."
        : "Setup stopped: a platform administrator already exists. No account was
changed.");
    return created ? 0 : 1;
}
}

```


PlatformAuthenticationResult.cs

```
using BridgeControl.DTOs.Platform;
using BridgeControl.Services.Shared;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Keeps the public login response separate from the refresh credential written to the secure
/// cookie.
/// </summary>
public sealed class PlatformAuthenticationResult(PlatformAuthSessionResponse response,
IssuedSession issued)
{
    public PlatformAuthSessionResponse Response { get; } = response;
    public string RefreshCredential { get; } = issued.RefreshCredential;
    public DateTimeOffset RefreshExpiresAt { get; } = issued.RefreshExpiresAt;
}
```

PlatformAuthenticationService.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Shared;
using BridgeControl.Options;
using BridgeControl.Security;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Authenticates platform administrators and connects shared session handling to platform
/// accounts and audit records.
/// </summary>
public class PlatformAuthenticationService(PlatformDbContext database,
    PasswordService<PlatformAdmin> passwords, AuthenticationSessionService sessions,
    PlatformAccessTokenService accessTokens, IOptions<PlatformAuthenticationOptions>
configuration,
    TimeProvider clock, ILogger<PlatformAuthenticationService> logger)
{
    /// <summary>
    /// Authenticates an active platform administrator and creates a session.
    /// </summary>
    public async Task<PlatformAuthenticationResult?> LoginAsync(string email, string password,
CancellationToken cancellationToken)
    {
        string normalizedEmail = AccountValidation.NormalizeEmail(email);
        PlatformAdmin? administrator = await database.PlatformAdmins
            .SingleOrDefaultAsync(entity => entity.Email == normalizedEmail,
cancellationToken);
        PasswordVerificationResult verification = passwords.Verify(administrator,
administrator?.PasswordHash, password);
        if (administrator is null || verification == PasswordVerificationResult.Failed
|| administrator.Status != PlatformAuthentication.ActiveStatus)
        {
            logger.LogWarning("Platform login failed.");
        }
    }
}
```

```

        if (administrator is not null)
        {
            if (administrator.Status != PlatformAuthentication.ActiveStatus)
            {
                await RevokeInactiveAccountAsync(administrator.Id,
cancellationToken);
            }
            RecordAccountEvent(administrator.Id, "LoginFailed");
            await database.SaveChangesAsync(cancellationToken);
        }
        return null;
    }

    if (verification == PasswordVerificationResult.SuccessRehashNeeded)
    {
        administrator.PasswordHash = passwords.Hash(administrator, password);
    }
    administrator.LastLoginAt = clock.GetUtcNow();
    PlatformAuthenticationOptions options = configuration.Value;
    IssuedSession issued = sessions.Create(administrator.Id,
        TimeSpan.FromDays(options.SessionDays),
        TimeSpan.FromDays(options.RefreshTokenDays));
    database.AuthenticationSessions.Add(issued.Session);
    RecordSessionEvent(issued.Session, "SessionCreated");
    await database.SaveChangesAsync(cancellationToken);
    return new PlatformAuthenticationResult(accessTokens.Create(administrator,
issued.Session), issued);
}

/// <summary>
/// Rotates the platform refresh credential after checking session and account validity.
/// </summary>
public async Task<PlatformAuthenticationResult?> RefreshAsync(string? credential,
CancellationToken cancellationToken)
{
    IssuedSession? issued = await sessions.RotateAsync(database, credential,
        TimeSpan.FromDays(configuration.Value.RefreshTokenDays),
AccountIsActiveAsync, RecordSessionEvent, cancellationToken);
    if (issued is null)
    {
        logger.LogWarning("Platform refresh failed.");
        return null;
    }
    PlatformAdmin administrator = await database.PlatformAdmins.AsNoTracking()
        .SingleAsync(entity => entity.Id == issued.Session.AccountId,
cancellationToken);
    if (administrator.Status != PlatformAuthentication.ActiveStatus)
    {
        await RevokeInactiveAccountAsync(administrator.Id, cancellationToken);
        return null;
    }
    return new PlatformAuthenticationResult(accessTokens.Create(administrator,
issued.Session), issued);
}

/// <summary>
/// Revokes the administrator's selected authentication session.
/// </summary>
public Task LogoutAsync(Guid sessionId, Guid accountId, CancellationToken
cancellationToken)
    => sessions.RevokeAsync(database, sessionId, accountId, RecordSessionEvent,
"SessionLoggedOut", cancellationToken);

```

```

    /// <summary>
    /// Checks that the platform account and session remain active.
    /// </summary>
    public async Task<bool> ValidateSessionAsync(Guid sessionId, Guid accountId,
CancellationTokens cancellationTokens)
    {
        if (!await AccountIsActiveAsync(accountId, cancellationTokens))
        {
            await RevokeInactiveAccountAsync(accountId, cancellationTokens);
            return false;
        }
        DateTimeOffset now = clock.UtcNow();
        return await database.AuthenticationSessions.AsNoTracking().AnyAsync(entity =>
            entity.Id == sessionId && entity.AccountId == accountId
            && entity.RevokedAt == null && entity.ExpiresAt > now, cancellationTokens);
    }

    /// <summary>
    /// Checks whether the platform administrator account is active.
    /// </summary>
    private async Task<bool> AccountIsActiveAsync(Guid accountId, CancellationTokens
cancellationTokens)
    {
        string? status = await database.PlatformAdmins.AsNoTracking()
            .Where(entity => entity.Id == accountId).Select(entity =>
entity.Status).SingleOrDefaultAsync(cancellationTokens);
        return status == PlatformAuthentication.ActiveStatus;
    }

    /// <summary>
    /// Revokes sessions that belong to an inactive platform administrator.
    /// </summary>
    private async Task RevokeInactiveAccountAsync(Guid accountId, CancellationTokens
cancellationTokens)
    {
        DateTimeOffset now = clock.UtcNow();
        int changed = await database.AuthenticationSessions.Where(entity =>
entity.AccountId == accountId && entity.RevokedAt == null)
            .ExecuteUpdateAsync(setters => setters.SetProperty(entity =>
entity.RevokedAt, now), cancellationTokens);
        if (changed > 0)
        {
            RecordAccountEvent(accountId, "AccountSessionsRevoked");
            await database.SaveChangesAsync(cancellationTokens);
        }
    }

    /// <summary>
    /// Adds a platform audit entry for an authentication session event.
    /// </summary>
    private void RecordSessionEvent(AuthenticationSession session, string action)
    {
        database.PlatformAuditLogs.Add(new PlatformAuditLog
        {
            PlatformAdminId = session.AccountId,
            Action = action,
            EntityType = nameof(AuthenticationSession),
            EntityId = session.Id.ToString(),
            Details = "{}",
            CreatedAt = clock.UtcNow()
        });
    }

```

```

    }

    /// <summary>
    /// Adds a platform audit entry for an administrator account event.
    /// </summary>
    private void RecordAccountEvent(Guid accountId, string action)
    {
        database.PlatformAuditLogs.Add(new PlatformAuditLog
        {
            PlatformAdminId = accountId,
            Action = action,
            EntityType = nameof(PlatformAdmin),
            EntityId = accountId.ToString(),
            Details = "{}",
            CreatedAt = clock.GetUtcNow()
        });
    }
}

```

PlatformVendorLock.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Serializes local platform vendor writes across API processes, including database/file
/// provisioning.
/// </summary>
public sealed class PlatformVendorLock : IAsyncDisposable
{
    private readonly SqlConnection connection;

    /// <summary>
    /// Initializes a new <see cref="PlatformVendorLock"/> instance and its dependencies.
    /// </summary>
    private PlatformVendorLock(SqlConnection connection) => this.connection = connection;

    /// <summary>
    /// Acquires the vendor-specific lock that serializes platform maintenance operations.
    /// </summary>
    public static async Task<PlatformVendorLock> AcquireAsync(PlatformDbContext database,
        CancellationToken cancellationToken)
    {
        // A dedicated non-pooled session releases the lock even after cancellation or a
        broken request.
        SqlConnectionStringBuilder settings = new(database.Database.GetConnectionString())
        { Pooling = false };
        SqlConnection connection = new(settings.ConnectionString);
        try
        {
            await connection.OpenAsync(cancellationToken);
            await using SqlCommand command = connection.CreateCommand();
            command.CommandText = "DECLARE @result int; EXEC @result =
sys.sp_getapplock @Resource = N'BridgeControl:VendorAdministration', @LockMode = 'Exclusive',
@LockOwner = 'Session', @LockTimeout = 15000; SELECT @result;";
            int result = (int) (await command.ExecuteScalarAsync(cancellationToken))!;

```

```

        if (result < 0)
        {
            throw new ApiConflictException("vendor-operation-busy", "Another
vendor operation is running. Try again shortly.");
        }
        return new PlatformVendorLock(connection);
    }
    catch
    {
        await connection.DisposeAsync();
        throw;
    }
}

/// <summary>
/// Asynchronously releases the resources owned by this instance.
/// </summary>
public ValueTask DisposeAsync() => connection.DisposeAsync();
}

```

SqlVendorMaintenance.cs

```

using BridgeControl.Data;
using BridgeControl.Options;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Migrations;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Native SQL backup/restore restricted to the infrastructure-resolved vendor and its existing
database files.
/// </summary>
public sealed class SqlVendorMaintenance(IOptions<DatabaseOperationsOptions> options) :
IVendorDatabaseMaintenance
{
    /// <summary>
    /// Applies pending vendor database migrations.
    /// </summary>
    public Task MigrateAsync(VendorDbContext database, string migration, CancellationToken ct)
=> database.GetService<IMigrator>().MigrateAsync(migration, ct);

    /// <summary>
    /// Creates a native SQL Server backup of the vendor database.
    /// </summary>
    public async Task BackupAsync(VendorDbContext database, string path, CancellationToken ct)
    {
        await using SqlConnection connection = Master(database); await
connection.OpenAsync(ct);
        await RunAsync(connection, $"BACKUP DATABASE {Name(database)} TO DISK = @path WITH
COPY_ONLY, CHECKSUM, INIT; RESTORE VERIFYONLY FROM DISK = @path WITH CHECKSUM;", path, ct);
    }

    /// <summary>
    /// Restores the vendor database from a native SQL Server backup.
    /// </summary>
}

```

```

        public async Task RestoreAsync(VendorDbContext database, string path, CancellationToken
ct)
        {
            await using SqlConnection connection = Master(database); await
connection.OpenAsync(ct);
            await RunAsync(connection, "RESTORE VERIFYONLY FROM DISK = @path WITH CHECKSUM;",
path, ct);
            using SqlCommand header = Command(connection, "RESTORE HEADERONLY FROM DISK =
@path;", path);
            await using (SqlDataReader reader = await header.ExecuteReaderAsync(ct))
            {
                if (!await reader.ReadAsync(ct) ||
reader.GetString(reader.GetOrdinal("DatabaseName")) !=
database.Database.GetDbConnection().Database
                || await reader.ReadAsync(ct)) { throw new
InvalidDataException("Backup does not identify the selected vendor database."); }
                Dictionary<int, string> targetFiles = [];
                using (SqlCommand files = Command(connection, "SELECT file_id, physical_name FROM
sys.master_files WHERE database_id = DB_ID(@name);"))
                {
                    files.Parameters.AddWithValue("@name",
database.Database.GetDbConnection().Database);
                    await using SqlDataReader reader = await files.ExecuteReaderAsync(ct);
                    while (await reader.ReadAsync(ct)) { targetFiles.Add(reader.GetInt32(0),
reader.GetString(1)); }
                }
                using SqlCommand restore = Command(connection, "", path);
                List<string> moves = [];
                using (SqlCommand listing = Command(connection, "RESTORE FILELISTONLY FROM DISK =
@path;", path))
                await using (SqlDataReader reader = await listing.ExecuteReaderAsync(ct))
                {
                    while (await reader.ReadAsync(ct))
                    {
                        int fileId =
checked((int)reader.GetInt64(reader.GetOrdinal("FileId")));
                        if (!targetFiles.Remove(fileId, out string? physical)) { throw new
InvalidDataException("Backup file layout does not match the selected database."); }
                        int index = moves.Count;
                        restore.Parameters.AddWithValue("@logical" + index,
reader.GetString(reader.GetOrdinal("LogicalName")));
                        restore.Parameters.AddWithValue("@physical" + index, physical);
                        moves.Add($"MOVE @logical{index} TO @physical{index}");
                    }
                }
                if (targetFiles.Count != 0 || moves.Count == 0) { throw new
InvalidDataException("Database file layout mismatch."); }
                SqlConnection.ClearPool((SqlConnection)database.Database.GetDbConnection());
                try
                {
                    await RunAsync(connection, $"ALTER DATABASE {Name(database)} SET
SINGLE_USER WITH ROLLBACK IMMEDIATE;", null, ct);
                    restore.CommandText = $"RESTORE DATABASE {Name(database)} FROM DISK = @path
WITH REPLACE, CHECKSUM, RECOVERY, {string.Join(", ", moves)}";
                    await restore.ExecuteNonQueryAsync(ct);
                }
                finally { await RunAsync(connection, $"ALTER DATABASE {Name(database)} SET
MULTI_USER;", null, CancellationToken.None); }
            }

            /// <summary>

```

```

    /// Checks that the restored vendor database can be used.
    /// </summary>
    public async Task VerifyAsync(VendorDbContext database, CancellationToken ct)
    {
        await using SqlConnection connection = Master(database); await
connection.OpenAsync(ct);
        using SqlCommand command = Command(connection, $"DBCC CHECKDB ({Name(database)})
WITH NO_INFOMSGS, ALL_ERRORMSG;");
        await using SqlDataReader reader = await command.ExecuteReaderAsync(ct);
        do { if (await reader.ReadAsync(ct)) { throw new InvalidDataException("Database
integrity verification failed."); } } while (await reader.NextResultAsync(ct));
    }

    /// <summary>
    /// Quotes a SQL Server database identifier for maintenance commands.
    /// </summary>
    private static string Name(VendorDbContext database)
    {
        string name = database.Database.GetDbConnection().Database;
        const string prefix = "BridgeControl_Vendor_";
        if (!name.StartsWith(prefix, StringComparison.Ordinal) ||
!Guid.TryParseExact(name[prefix.Length..], "N", out Guid id) || name !=
VendorInfrastructure.DatabaseName(id))
        { throw new InvalidOperationException("Untrusted vendor database name."); }
        return "[" + name + "]";
    }

    /// <summary>
    /// Creates a connection to the SQL Server master database.
    /// </summary>
    private static SqlConnection Master(VendorDbContext database)
    {
        _ = Name(database);
        return new SqlConnection(new
SqlConnectionStringBuilder(database.Database.GetConnectionString()) { InitialCatalog = "master",
Pooling = false }.ConnectionString);
    }

    /// <summary>
    /// Creates a maintenance command with its configured timeout.
    /// </summary>
    private SqlCommand Command(SqlConnection connection, string sql, string? path = null)
    {
        SqlCommand command = connection.CreateCommand(); command.CommandText = sql;
        command.CommandTimeout = Math.Clamp(options.Value.SqlTimeoutSeconds, 30, 3600);
        if (path is not null) { command.Parameters.AddWithValue("@path", path); }
        return command;
    }

    /// <summary>
    /// Executes a SQL Server maintenance command.
    /// </summary>
    private async Task RunAsync(SqlConnection connection, string sql, string? path,
CancellationToken ct)
    {
        using SqlCommand command = Command(connection, sql, path); await
command.ExecuteNonQueryAsync(ct); }
    }

```

VendorBackupStore.cs

```

using System.IO.Compression;
using System.Security.Cryptography;
using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Represents backup artifact data.
/// </summary>
public sealed record BackupArtifact(Guid Id, string StorageIdentifier, long SizeBytes, string
Sha256);
/// <summary>
/// Represents backup manifest data.
/// </summary>
public sealed record BackupManifest(int Format, Guid VendorId, Guid BackupId, string Schema,
BackupArtifact[] Artifacts);

/// <summary>Pairs an offline native SQL backup with verified immutable blobs in an authenticated
encrypted bundle.</summary>
public sealed class VendorBackupStore(IOptions<DatabaseOperationsOptions> settings,
IVendorInfrastructure infrastructure,
    IVendorDatabaseMaintenance sql, VendorArtifactArchive archive)
{
    private DatabaseOperationsOptions Options => settings.Value;

    /// <summary>
    /// Resolves the storage root for a vendor's backups.
    /// </summary>
    public string Root()
    {
        if (Options.MaximumBundleBytes < 1024 * 1024 || Options.MaximumArtifacts is < 1 or
> 1000000 || Options.RetentionDays is < 1 or > 3650)
        { throw new InvalidOperationException("Invalid backup limits."); }
        return PrivateDirectory.Ensure(Options.BackupRoot);
    }

    /// <summary>
    /// Resolves the final directory for a backup operation.
    /// </summary>
    public string Bundle(Guid id) => Path.Combine(Root(), id.ToString("N") + ".bcbackup");

    /// <summary>
    /// Resolves the temporary directory for a backup operation.
    /// </summary>
    public string Stage(Guid id) => PrivateDirectory.Ensure(Path.Combine(Root(),
id.ToString("N") + ".stage"), Options.SqlServiceSid);

    /// <summary>
    /// Bundles and encrypts database and artifact data with a backup manifest.
    /// </summary>
    public async Task<string> CreateAsync(SoftwareVendor vendor, VendorDbContext database,
Guid id, string schema, CancellationToken ct)
    {

```



```

        if (!Options.EncryptionKeys.TryGetValue(Options.ActiveKeyId, out string? secret)
|| Convert.FromBase64String(secret).Length != 32)
        { throw new InvalidOperationException("Configure a 256-bit backup encryption key
before backing up."); }
        string stage = Stage(id); CleanupStage(id);
        string native = Path.Combine(stage, "database.bak"), zipPath = Path.Combine(stage,
"bundle.zip"), encrypted = Path.Combine(stage, "bundle.encrypted");
        try
        {
            FileArtifact[] artifacts = await
database.FileArtifacts.AsNoTracking().Take(Options.MaximumArtifacts + 1).ToArrayAsync(ct);
            if (artifacts.Length > Options.MaximumArtifacts || artifacts.Select(x =>
x.StorageIdentifier).Distinct().Count() != artifacts.Length)
            { throw new InvalidDataException("Archive metadata exceeds backup limits or
contains duplicate storage identifiers."); }
            long allocated = await database.Database.SqlQueryRaw<long>("SELECT
SUM(CONVERT(bigint, size)) * 8192 AS [Value] FROM sys.database_files").SingleAsync(ct);
            long estimated = checked(allocated + artifacts.Sum(x => x.SizeBytes) + 16L
* 1024 * 1024);
            CheckSize(estimated);
            if (new DriveInfo(Path.GetPathRoot(stage)!).AvailableFreeSpace < checked(3
* estimated))
            { throw new IOException("Insufficient free staging space for a consistent
encrypted backup."); }
            await sql.BackupAsync(database, native, ct);
            long total = new FileInfo(native).Length;
            await using (FileStream output = new(zipPath, FileMode.CreateNew,
FileAccess.ReadWrite, FileShare.None))
            using (ZipArchive zip = new(output, ZipArchiveMode.Create, leaveOpen:
true))
            {
                BackupManifest manifest = new(1, vendor.Id, id, schema,
artifacts.Select(x => new BackupArtifact(x.Id, x.StorageIdentifier, x.SizeBytes,
x.Sha256)).ToArray());
                await using (Stream entry =
zip.CreateEntry("manifest.json").Open()) { await JsonSerializer.SerializeAsync(entry, manifest,
cancellationTokens: ct); }
                await using (FileStream source = File.OpenRead(native))
                await using (Stream entry = zip.CreateEntry("database.bak",
CompressionLevel.NoCompression).Open())
                { CheckSize(total); await source.CopyToAsync(entry, ct); }
                foreach (FileArtifact artifact in artifacts)
                {
                    total = checked(total + artifact.SizeBytes);
                    CheckSize(total);
                    await using Stream source = await
archive.OpenAsync(infrastructure.GetArchivePath(vendor), artifact, ct);
                    await using Stream entry = zip.CreateEntry("blobs/" +
artifact.StorageIdentifier + ".blob", CompressionLevel.NoCompression).Open();
                    await source.CopyToAsync(entry, ct);
                }
            }
            await EncryptedBackup.EncryptAsync(zipPath, encrypted, Options.ActiveKeyId,
Convert.FromBase64String(secret), Options.MaximumBundleBytes, ct);
            // Retrying an interrupted publication replaces only this operation's
generated bundle.
            RejectLink(Bundle(id)); File.Move(encrypted, Bundle(id), overwrite: true);
            return await HashAsync(Bundle(id), ct);
        }
        finally { CleanupStage(id); }
    }
}

```

```

    /// <summary>
    /// Verifies and decrypts a backup into a restore staging directory.
    /// </summary>
    public async Task<BackupManifest> PrepareRestoreAsync(Guid vendorId, Guid operationId,
VendorMaintenanceOperation backup, CancellationToken ct)
    {
        if (backup.VendorId != vendorId || backup.Kind != "Backup" || backup.Status !=
"Succeeded" || backup.ExpiredAt is not null
            || await HashAsync(Bundle(backup.Id), ct) != backup.BundleSha256) { throw
new InvalidDataException("Backup is unavailable or does not match the selected vendor."); }
        string stage = Stage(operationId); CleanupStage(operationId);
        string zipPath = Path.Combine(stage, "bundle.zip");
        await EncryptedBackup.DecryptAsync(Bundle(backup.Id), zipPath,
Options.EncryptionKeys, Options.MaximumBundleBytes, ct);
        using ZipArchive zip = ZipFile.OpenRead(zipPath);
        if (zip.Entries.Count > Options.MaximumArtifacts + 2 || zip.Entries.Select(x =>
x.FullName).Distinct(StringComparer.Ordinal).Count() != zip.Entries.Count)
        { throw new InvalidDataException("Invalid backup entries."); }
        ZipArchiveEntry manifestEntry = zip.GetEntry("manifest.json") ?? throw new
InvalidDataException("Backup manifest missing.");
        if (manifestEntry.Length > 32 * 1024 * 1024) { throw new
InvalidDataException("Backup manifest exceeds limit."); }
        BackupManifest manifest;
        await using (Stream entry = manifestEntry.Open()) { manifest = await
JsonSerializer.DeserializeAsync<BackupManifest>(entry, cancellationToken: ct) ?? throw new
InvalidDataException("Invalid manifest."); }
        if (manifest.Format != 1 || manifest.VendorId != vendorId || manifest.BackupId !=
backup.Id || manifest.Schema != backup.CurrentSchema
            || manifest.Artifacts.Length > Options.MaximumArtifacts ||
manifest.Artifacts.Select(x => x.Id).Distinct().Count() != manifest.Artifacts.Length)
        { throw new InvalidDataException("Backup manifest does not match its journal."); }
        Dictionary<string, BackupArtifact> expected = new(StringComparer.Ordinal);
        foreach (BackupArtifact artifact in manifest.Artifacts)
        {
            if (!Guid.TryParseExact(artifact.StorageIdentifier, "N", out Guid storage)
|| storage.ToString("N") != artifact.StorageIdentifier || artifact.SizeBytes < 0)
            { throw new InvalidDataException("Invalid artifact metadata."); }
            expected.Add("blobs/" + artifact.StorageIdentifier + ".blob", artifact);
        }
        if (zip.Entries.Count != expected.Count + 2 || zip.GetEntry("database.bak") is
null) { throw new InvalidDataException("Incomplete backup."); }
        long total = 0;
        foreach (ZipArchiveEntry entry in zip.Entries)
        {
            if (entry.FullName == "manifest.json") { continue; }
            bool database = entry.FullName == "database.bak";
            if (!database && !expected.ContainsKey(entry.FullName)) { throw new
InvalidDataException("Unexpected backup entry."); }
            total = checked(total + entry.Length); CheckSize(total);
            string destination = database ? Path.Combine(stage, "database.bak") :
Path.Combine(stage, expected[entry.FullName].StorageIdentifier + ".blob");
            await using (Stream source = entry.Open())
            await using (FileStream output = new(destination, FileMode.CreateNew,
FileAccess.Write, FileShare.None))
            {
                byte[] buffer = new byte[81920]; long written = 0; int count;
                while ((count = await source.ReadAsync(buffer, ct)) != 0)
                { written += count; if (written > entry.Length) { throw new
InvalidDataException("Backup entry exceeds declared size."); } await
output.WriteAsync(buffer.AsMemory(0, count), ct); }
                if (written != entry.Length) { throw new
InvalidDataException("Truncated backup entry."); }
            }
        }
    }

```

```

        }
        if (!database && (entry.Length != expected[entry.FullName].SizeBytes ||
await HashAsync(destination, ct) != expected[entry.FullName].Sha256))
        { throw new InvalidDataException("Backup artifact integrity check
failed."); }
    }
    return manifest;
}

/// <summary>
/// Publishes staged backup data after verifying its recorded integrity.
/// </summary>
public async Task PublishAndVerifyAsync(SoftwareVendor vendor, VendorDbContext database,
Guid operationId, BackupManifest manifest, CancellationToken ct)
{
    FileArtifact[] actual = await
database.FileArtifacts.AsNoTracking().ToArrayAsync(ct);
    if (!actual.Select(x => new BackupArtifact(x.Id, x.StorageIdentifier, x.SizeBytes,
x.Sha256)).OrderBy(x => x.Id).SequenceEqual(manifest.Artifacts.OrderBy(x => x.Id)))
    { throw new InvalidDataException("Restored database and archive manifest
differ."); }
    string blobs =
PrivateDirectory.Ensure(Path.Combine(infrastructure.GetArchivePath(vendor), "blobs"));
    foreach (FileArtifact artifact in actual)
    {
        string target = Path.Combine(blobs, artifact.StorageIdentifier + ".blob");
RejectLink(target);
        string temporary = Path.Combine(blobs, operationId.ToString("N") +
".restoring"); RejectLink(temporary);
        await using (FileStream source =
File.OpenRead(Path.Combine(Stage(operationId), artifact.StorageIdentifier + ".blob")))
        await using (FileStream output = new(temporary, FileMode.Create,
FileAccess.Write, FileShare.None))
        { await source.CopyToAsync(output, ct); await output.FlushAsync(ct);
output.Flush(true); }
        File.Move(temporary, target, overwrite: true);
        await using Stream verified = await
archive.OpenAsync(infrastructure.GetArchivePath(vendor), artifact, ct);
    }
}

/// <summary>
/// Removes the temporary files for a completed or abandoned backup stage.
/// </summary>
public void CleanupStage(Guid id)
{
    string stage = Stage(id);
    // No recursive traversal: only our fixed filenames and canonical blob names are
eligible.
    foreach (string path in Directory.EnumerateFiles(stage))
    {
        string name = Path.GetFileName(path);
        if (name is not ("database.bak" or "bundle.zip" or "bundle.encrypted")
&& !(Path.GetExtension(name) == ".blob" &&
Guid.TryParseExact(Path.GetFileNameWithoutExtension(name), "N", out _)))
        { throw new InvalidDataException("Unexpected staging file; operator
inspection required."); }
        RejectLink(path); File.Delete(path);
    }
}

/// <summary>

```

```

    /// Computes the SHA-256 digest of a backup file.
    /// </summary>
    public static async Task<string> HashAsync(string path, CancellationToken ct)
    { RejectLink(path); await using FileStream file = File.OpenRead(path); return
Convert.ToHexStringLower(await SHA256.HashDataAsync(file, ct)); }

    /// <summary>
    /// Rejects a backup file whose length differs from its manifest entry.
    /// </summary>
    private void CheckSize(long size) { if (size > Options.MaximumBundleBytes) { throw new
InvalidDataException("Backup exceeds configured storage limit."); } }

    /// <summary>
    /// Rejects filesystem links in backup paths.
    /// </summary>
    public static void RejectLink(string path)
    { if (File.Exists(path) && (File.GetAttributes(path) & FileAttributes.ReparsePoint) != 0)
{ throw new InvalidDataException("Backup storage cannot contain links."); } }
}

```

VendorDatabaseOperations.cs

```

using System.Text.Json;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Options;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>Offline, resumable vendor maintenance. Failures reserve the vendor until the same
operation succeeds.</summary>
public sealed class VendorDatabaseOperations(PlatformDbContext platform, IVendorInfrastructure
infrastructure,
    IVendorDatabaseMaintenance sql, VendorBackupStore backups,
    IOptions<DatabaseOperationsOptions> options, TimeProvider clock)
{
    /// <summary>
    /// Runs or resumes a durable vendor database maintenance operation.
    /// </summary>
    public async Task<VendorMaintenanceOperation> ExecuteAsync(Guid administratorId, Guid
vendorId, Guid operationId, string kind, Guid? backupId = null, CancellationToken ct = default)
    {
        if (operationId == Guid.Empty || kind is not ("Upgrade" or "Backup" or "Restore")
|| (kind == "Restore") != backupId.HasValue)
        { throw new ArgumentException("Choose Upgrade, Backup or Restore and a nonempty
operation ID; only Restore requires a backup ID."); }
        await using DatabaseHostLease offline = await
DatabaseHostLease.AcquireAsync(platform, ct);
        await using PlatformVendorLock serialization = await
PlatformVendorLock.AcquireAsync(platform, ct);
        if (!await platform.PlatformAdmins.AnyAsync(x => x.Id == administratorId &&
x.Status == "Active", ct))
        { throw new ArgumentException("An active platform administrator is required."); }
        SoftwareVendor vendor = await platform.SoftwareVendors.Include(x =>
x.Database).Include(x => x.FileArchive).Include(x => x.Provisioning).SingleAsync(x => x.Id ==
vendorId, ct);
        await using VendorDbContext database = infrastructure.OpenDatabase(vendor);

```

```

        string[] known = database.Database.GetMigrations().ToArray();
        VendorMaintenanceOperation? selectedBackup = null;
        if (kind == "Restore")
        {
            selectedBackup = await
platform.VendorMaintenanceOperations.AsNoTracking().SingleOrDefaultAsync(x => x.Id == backupId,
ct);

            if (selectedBackup is null || selectedBackup.VendorId != vendorId ||
selectedBackup.Kind != "Backup" || selectedBackup.Status != "Succeeded"
|| selectedBackup.ExpiredAt is not null ||
selectedBackup.CurrentSchema != known.Last())
            { throw new InvalidDataException("Select a completed, retained backup for
this vendor and schema release."); }
        }
        VendorMaintenanceOperation? operation = await
platform.VendorMaintenanceOperations.SingleOrDefaultAsync(x => x.Id == operationId, ct);
        if (operation is not null)
        {
            if (operation.VendorId != vendorId || operation.Kind != kind ||
operation.BackupId != backupId || operation.TargetSchema != known.Last())
            { throw new ArgumentException("Operation identity or software version does
not match the journal. Resume using the original release."); }
            if (operation.Status == "Succeeded") { return operation; }
        }
        else
        {
            if (vendor.Database?.Status != "Ready" || vendor.FileArchive?.Status !=
"Ready" || vendor.Provisioning?.SeededAt is null
|| await platform.VendorMaintenanceOperations.AnyAsync(x =>
x.VendorId == vendorId && x.IsActive, ct))
            { throw new ArgumentException("The vendor is not ready or already has an
unfinished operation. Resume that operation first."); }
            // A new restore must also work when SQL data is damaged and its migration
table is unreadable.
            string[] applied = kind == "Restore" ? [] : (await
database.Database.GetAppliedMigrationsAsync(ct)).ToArray();
            if (kind != "Restore") { ValidateHistory(known, applied); }
            if (kind == "Backup" && applied.LastOrDefault() != known.Last())
            { throw new ArgumentException("Upgrade this vendor before creating a backup
with this release."); }
            string sourceSchema = kind == "Restore" ?
vendor.Provisioning!.SchemaVersion : applied.LastOrDefault() ?? "";
            operation = new()
            {
                Id = operationId,
                VendorId = vendorId,
                PlatformAdminId = administratorId,
                Kind = kind,
                BackupId = backupId,
                SourceSchema = sourceSchema,
                CurrentSchema = sourceSchema,
                TargetSchema = known.Last(),
                OperatorAccount = Environment.UserDomainName + "\\\" +
Environment.UserName,
                OperatorHost = Environment.MachineName,
                CreatedAt = clock.GetUtcNow(),
                UpdatedAt = clock.GetUtcNow()
            };
            platform.VendorMaintenanceOperations.Add(operation);
        }
        operation.Status = "Running"; operation.FailureCode = null;
        vendor.Database!.Status = "Maintenance";

```

```

        await SaveAsync(operation, administratorId, "Started", ct);
    try
    {
        string[] applied = kind == "Restore" ? [] : (await
database.Database.GetAppliedMigrationsAsync(ct)).ToArray();
        if (kind != "Restore") { ValidateHistory(known, applied); }
        if (kind == "Upgrade")
        {
            foreach (string migration in known.Skip(applied.Length))
            {
                await offline.CheckAsync(ct);
                await SaveAsync(operation, administratorId, migration, ct);
                await sql.MigrateAsync(database, migration, ct);
                operation.CurrentSchema = migration;
                await SaveAsync(operation, administratorId,
"MigrationApplied", ct);
            }
        }
        else if (kind == "Backup")
        {
            if (applied.LastOrDefault() != known.Last()) { throw new
InvalidOperationException("Upgrade this vendor before creating a backup."); }
            await SaveAsync(operation, administratorId, "BackingUp", ct);
            operation.BundleSha256 = await backups.CreateAsync(vendor,
database, operation.Id, operation.CurrentSchema, ct);
        }
        else
        {
            await SaveAsync(operation, administratorId, "ValidatingBackup",
ct);

            BackupManifest manifest = await
backups.PrepareRestoreAsync(vendorId, operationId, selectedBackup!, ct);
            // Restore only a same-release snapshot. Upgrade older snapshots
under their original release first.
            if (manifest.Schema != known.Last()) { throw new
InvalidDataException("Restore requires a backup from the current schema release."); }
            await SaveAsync(operation, administratorId, "RestoringDatabase",
ct);

            await offline.CheckAsync(ct);
            await sql.RestoreAsync(database,
Path.Combine(backups.Stage(operationId), "database.bak"), ct);
            await SaveAsync(operation, administratorId, "RestoringArchive",
ct);

            await backups.PublishAndVerifyAsync(vendor, database, operationId,
manifest, ct);

            // Platform cutoff survives restoration and invalidates restored
sessions and Agent credentials.
            vendor.SessionsRevokedBefore = clock.UtcNow();
            await RevokeRestoredCredentialsAsync(database,
vendor.SessionsRevokedBefore.Value, ct);
            backups.CleanupStage(operationId);
        }
        await SaveAsync(operation, administratorId, "Verifying", ct);
        await sql.VerifyAsync(database, ct);
        string[] final = (await
database.Database.GetAppliedMigrationsAsync(ct)).ToArray();
        if (!final.SequenceEqual(known)) { throw new InvalidDataException("Vendor
schema verification failed."); }
        await offline.CheckAsync(ct);
        operation.CurrentSchema = known.Last(); operation.Status = "Succeeded";
operation.IsActive = false; operation.CompletedAt = clock.UtcNow();

```

```

        vendor.Database.Status = "Ready"; vendor.Provisioning!.SchemaVersion =
known.Last(); vendor.Provisioning.UpdatedAt = clock.UtcNow();
        await SaveAsync(operation, administratorId, "Completed", ct);
        return operation;
    }
    catch
    {
        operation.Status = "Failed"; operation.IsActive = true;
operation.CompletedAt = null; vendor.Database.Status = "Maintenance";
        operation.FailureCode = "vendor-maintenance-failed";
        // Preserve the failed stage; only safe codes go to journal/log output.
        await SaveAsync(operation, administratorId, operation.Stage,
CancellationToken.None);
        throw;
    }
    finally { if (kind == "Restore") { backups.CleanupStage(operationId); } }
}

/// <summary>
/// Schedules and runs vendor backups that are due.
/// </summary>
public async Task<int> BackupDueAsync(Guid administratorId, Cancellation token ct =
default)
{
    Guid[] vendors = await platform.SoftwareVendors.AsNoTracking().Where(x =>
x.Database != null && (x.Database.Status == "Ready" || x.Database.Status == "Maintenance"))
        .OrderBy(x => x.Id).Select(x => x.Id).ToArrayAsync(ct);
    int failures = 0;
    foreach (Guid vendorId in vendors)
    {
        platform.ChangeTracker.Clear();
        VendorMaintenanceOperation? active = await
platform.VendorMaintenanceOperations.AsNoTracking().SingleOrDefaultAsync(x => x.VendorId ==
vendorId && x.IsActive, ct);
        if (active is not null && active.Kind != "Backup") { failures++; continue;
}

        DateTimeOffset dayStart = new(clock.UtcNow().UtcDateTime.Date,
(TimeSpan.Zero);

        if (active is null && await platform.VendorMaintenanceOperations.AnyAsync(x
=> x.VendorId == vendorId && x.Kind == "Backup" && x.Status == "Succeeded" && x.CompletedAt >=
dayStart, ct)) { continue; }
        try { await ExecuteAsync(administratorId, vendorId, active?.Id ??
Guid.NewGuid(), "Backup", ct: ct); }
        catch (OperationCanceledException) when (ct.IsCancellationRequested) {
throw; }

        catch { failures++; }
    }
    return failures;
}

/// <summary>
/// Removes expired backups while preserving those required by active operations.
/// </summary>
public async Task ExpireBackupsAsync(Guid administratorId, Cancellation token ct = default)
{
    await using DatabaseHostLease offline = await
DatabaseHostLease.AcquireAsync(platform, ct);
    await using PlatformVendorLock serialization = await
PlatformVendorLock.AcquireAsync(platform, ct);
    if (!await platform.PlatformAdmins.AnyAsync(x => x.Id == administratorId &&
x.Status == "Active", ct)) { throw new ArgumentException("An active platform administrator is
required."); }
}

```

```

        DateTimeOffset cutoff = clock.GetUtcNow().AddDays(-options.Value.RetentionDays);
        VendorMaintenanceOperation[] expired = await
platform.VendorMaintenanceOperations.Where(x => x.Kind == "Backup" && x.Status == "Succeeded"
        && x.CompletedAt < cutoff && x.ExpiredAt == null &&
!platform.VendorMaintenanceOperations.Any(r => r.IsActive && r.BackupId ==
x.Id)).Take(1000).ToArrayAsync(ct);
        foreach (VendorMaintenanceOperation backup in expired)
        {
            string path = backups.Bundle(backup.Id);
VendorBackupStore.RejectLink(path); File.Delete(path);
            backup.ExpiredAt = clock.GetUtcNow(); await SaveAsync(backup,
administratorId, "Expired", ct);
        }
    }

    /// <summary>
    /// Persists maintenance operation progress.
    /// </summary>
    private async Task SaveAsync(VendorMaintenanceOperation operation, Guid administratorId,
string stage, CancellationToken ct)
    {
        operation.Stage = stage; operation.UpdatedAt = clock.GetUtcNow();
platform.PlatformAuditLogs.Add(new()
    {
        Id = Guid.NewGuid(),
        PlatformAdminId = administratorId,
        VendorId = operation.VendorId,
        Action = "VendorMaintenance" + operation.Status,
        EntityType = "VendorMaintenanceOperation",
        EntityId = operation.Id.ToString(),
        CreatedAt = operation.UpdatedAt,
        Details = JsonSerializer.Serialize(new
        {
            operation.Kind,
            Stage = stage,
            operation.FailureCode,
            operation.CurrentSchema,
            operation.TargetSchema,
            OperatorAccount = Environment.UserDomainName + "\\\" +
Environment.UserName,
            OperatorHost = Environment.MachineName
        })
    });
        await platform.SaveChangesAsync(ct);
    }

    /// <summary>
    /// Checks recorded migration history before database maintenance proceeds.
    /// </summary>
    private static void ValidateHistory(string[] known, string[] applied)
    { if (applied.Length > known.Length || !applied.SequenceEqual(known.Take(applied.Length)))
    { throw new InvalidDataException("Unrecognized or nonsequential vendor migration history."); } }

    /// <summary>
    /// Invalidates credentials restored from a backup before the vendor resumes service.
    /// </summary>
    private static async Task RevokeRestoredCredentialsAsync(VendorDbContext database,
DateTimeOffset cutoff, CancellationToken ct)
    {
        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(ct);

```



```

        await database.AuthenticationSessions.Where(x => x.RevokedAt ==
null).ExecuteUpdateAsync(x => x.SetProperty(y => y.RevokedAt, cutoff), ct);
        await database.AgentCredentials.Where(x => x.RevokedAt ==
null).ExecuteUpdateAsync(x => x.SetProperty(y => y.RevokedAt, cutoff), ct);
        await database.AgentEnrollments.Where(x => x.RevokedAt == null && x.UsedAt ==
null).ExecuteUpdateAsync(x => x.SetProperty(y => y.RevokedAt, cutoff), ct);
        await database.AccountActivations.Where(x => x.RevokedAt == null && x.UsedAt ==
null)
            .ExecuteUpdateAsync(x => x.SetProperty(y => y.RevokedAt,
cutoff).SetProperty(y => y.ProtectedToken, (string?)null), ct);
        await database.PasswordResets.Where(x => x.RevokedAt == null && x.UsedAt == null)
            .ExecuteUpdateAsync(x => x.SetProperty(y => y.RevokedAt,
cutoff).SetProperty(y => y.ProtectedToken, (string?)null), ct);
        await transaction.CommitAsync(ct);
    }
}

```

VendorInfrastructure.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Uses configured SQL Server credentials and an OS-private archive root without accepting
client storage paths.
/// </summary>
public class VendorInfrastructure(IConfiguration configuration,
IOptions<VendorProvisioningOptions> options) : IVendorInfrastructure
{
    public const string ConnectionReference = "ConnectionStrings:VendorServer";
    public const string ArchiveReference = "LocalServiceIdentity";
    /// <summary>
    /// Derives the vendor database name from its stable identifier.
    /// </summary>
    public static string DatabaseName(Guid vendorId) => $"BridgeControl_Vendor_{vendorId:N}";

    /// <summary>
    /// Opens a context for the vendor database.
    /// </summary>
    public VendorDbContext OpenDatabase(SoftwareVendor vendor)
    {
        if (vendor.Id == Guid.Empty || vendor.Database?.DatabaseName !=
DatabaseName(vendor.Id)
            || vendor.Database.ConnectionSecretRef != ConnectionReference)
        {
            throw new InvalidOperationException("The vendor database reference is
invalid.");
        }
        string connection = configuration.GetConnectionString("VendorServer")
?? configuration.GetConnectionString("PlatformDatabase")
?? throw new InvalidOperationException("The vendor SQL Server connection is
missing.");
    }
}

```

```

        SqlConnectionStringBuilder settings = new(connection) { InitialCatalog =
vendor.Database.DatabaseName };
        return new VendorDbContext(new
DbContextOptionsBuilder<VendorDbContext>().UseSqlServer(settings.ConnectionString).Options);
    }

    /// <summary>
    /// Resolves the vendor-specific artifact archive path.
    /// </summary>
    public string GetArchivePath(SoftwareVendor vendor)
    {
        if (vendor.Id == Guid.Empty || vendor.FileArchive?.StorageIdentifier !=
vendor.Id.ToString("N")
            || vendor.FileArchive.AccessSecretRef != ArchiveReference ||
!Path.IsPathFullyQualified(options.Value.StorageRoot))
        {
            throw new InvalidOperationException("The vendor archive reference is
invalid.");
        }
        return Path.Combine(Path.GetFullPath(options.Value.StorageRoot),
vendor.Id.ToString("N"));
    }

    /// <summary>
    /// Creates and protects the vendor artifact archive directory.
    /// </summary>
    public void EnsureArchive(SoftwareVendor vendor)
    {
        PrivateDirectory.Ensure(options.Value.StorageRoot);
        PrivateDirectory.Ensure(GetArchivePath(vendor));
    }
}

```

VendorInvitationService.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Platform;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Shared;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Establishes the initial vendor administrator and safely retries its private invitation
delivery.
/// </summary>
public class VendorInvitationService(PlatformDbContext platform, IVendorInfrastructure
infrastructure,
    AccountInvitationService accountInvitations, TimeProvider clock)
{
    /// <summary>
    /// Creates or renews the initial vendor administrator invitation.
    /// </summary>

```

```

        public async Task<UserResponse?> InviteInitialAsync(Guid vendorId,
CreateVendorAdministratorRequest request,
        Guid administratorId, CancellationToken cancellationToken)
        {
            await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, cancellationToken);
            SoftwareVendor? vendor = await platform.SoftwareVendors.Include(vendor =>
vendor.Database)
                .Include(vendor => vendor.FileArchive).Include(vendor =>
vendor.Provisioning)
                .SingleOrDefaultAsync(vendor => vendor.Id == vendorId, cancellationToken);
            if (vendor is null) { return null; }
            if (vendor.Status is not ("AwaitingAdministrator" or "Active")) ||
vendor.Database?.Status != "Ready"
                || vendor.FileArchive?.Status != "Ready" || vendor.Provisioning?.SeededAt
is null)
            {
                throw new ApiConflictException("vendor-not-ready", "Finish vendor
provisioning before inviting its administrator.");
            }
            string email = AccountValidation.NormalizeEmail(request.Email);
            await using VendorDbContext database = infrastructure.OpenDatabase(vendor);
            UserAccount user;
            AccountActivation? activation = null;
            await using (IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(cancellationToken))
            {
                Role role = await database.Roles.SingleAsync(role => role.Name ==
VendorRoleSeeder.Administrator, cancellationToken);
                List<UserAccount> administrators = await
database.UserAccounts.Where(account =>
                    account.UserRoles.Any(grant => grant.RoleId ==
role.Id)).Take(2).ToListAsync(cancellationToken);
                if (administrators.Count > 1 || (administrators.Count == 0 &&
vendor.Provisioning!.InvitationEstablishedAt is not null))
                {
                    throw new ApiConflictException("initial-administrator-exists", "The
initial administrator has already been established.");
                }
                UserAccount? existing = administrators.SingleOrDefault();
                if (existing is not null && (existing.Email != email || existing.Status is
not ("Invited" or "Active")))
                {
                    throw new ApiConflictException("initial-administrator-exists", "The
initial administrator has already been established.");
                }
                user = existing ?? new UserAccount
                {
                    Id = Guid.NewGuid(),
                    Email = email,
                    Status = "Invited",
                    CreatedAt = clock.GetUtcNow(),
                    UserRoles = [new UserRole { Id = Guid.NewGuid(), RoleId = role.Id
}]}
                };
                if (existing is null) { database.UserAccounts.Add(user); }
                if (user.Status == "Invited") { activation = await
accountInvitations.EnsureAsync(database, vendor.Id, user, administratorId,
"InitialAdministratorInvited", "PlatformAdmin", cancellationToken); }
                await database.SaveChangesAsync(cancellationToken);
                await transaction.CommitAsync(cancellationToken);
            }
        }
    }

```

```

        try
        {
            if (activation is not null) { await
accountInvitations.DeliverAsync(database, vendor.Id, user, activation, cancellationTokens); }
        }
        catch
        {
            vendor.Provisioning!.FailureCode = "invitation-delivery-failed";
            vendor.Provisioning.UpdatedAt = clock.UtcNow();
            platform.PlatformAuditLogs.Add(new PlatformAuditLog
            {
                PlatformAdminId = administratorId,
                VendorId = vendor.Id,
                Action = "InitialVendorInvitationDeliveryFailed",
                EntityType = nameof(SoftwareVendor),
                EntityId = vendor.Id.ToString(),
                Details = "{}",
                CreatedAt = clock.UtcNow()
            });
            await platform.SaveChangesAsync(CancellationTokens.None);
            throw;
        }
        if (vendor.Provisioning!.InvitationEstablishedAt is null ||
vendor.Provisioning.FailureCode is not null)
        {
            vendor.Provisioning.InvitationEstablishedAt ??= clock.UtcNow();
            vendor.Provisioning.FailureCode = null;
            vendor.Provisioning.UpdatedAt = clock.UtcNow();
            vendor.Status = "Active";
            platform.PlatformAuditLogs.Add(new PlatformAuditLog
            {
                PlatformAdminId = administratorId,
                VendorId = vendor.Id,
                Action = "InitialVendorAdministratorEstablished",
                EntityType = nameof(SoftwareVendor),
                EntityId = vendor.Id.ToString(),
                Details = "{}",
                CreatedAt = clock.UtcNow()
            });
            await platform.SaveChangesAsync(cancellationTokens);
        }
        return new UserResponse { Id = user.Id, CustomerId = user.CustomerId, Email =
user.Email, Status = user.Status, CreatedAt = user.CreatedAt, Roles =
[VendorRoleSeeder.Administrator] };
    }
}

```

VendorLifecycleService.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Platform;

/// <summary>Disables centrally before cleanup; reactivation finishes cleanup before allowing
access.</summary>

```

```

public class VendorLifecycleService(PlatformDbContext platform, IVendorInfrastructure
infrastructure,
    VendorSecurityAudit audit, TimeProvider clock)
{
    /// <summary>
    /// Changes vendor lifecycle status and applies the associated access revocation.
    /// </summary>
    public async Task<int> SetActiveAsync(Guid vendorId, Guid actorId, bool active,
CancellationTokens ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        SoftwareVendor? vendor = await platform.SoftwareVendors.Include(x =>
x.Database).Include(x => x.FileArchive).Include(x => x.Provisioning)
        .SingleOrDefaultAsync(x => x.Id == vendorId, ct);
        if (vendor is null) { return 404; }
        if (vendor.Database?.Status == "Maintenance") { return 409; }
        if (vendor.Status is not ("Active" or "Disabled") ||
vendor.Provisioning?.InvitationEstablishedAt is null) { return 409; }
        bool changed = vendor.Status != (active ? "Active" : "Disabled");
        if (!active)
        {
            vendor.SessionsRevokedBefore = clock.GetUtcNow();
            vendor.Status = "Disabled"; vendor.DeactivatedAt ??= clock.GetUtcNow();
            PlatformAudit(vendorId, actorId, "VendorDeactivated", changed);
            // This commit is the authoritative access gate; it remains closed if
cleanup fails.
            await platform.SaveChangesAsync(ct);
        }
        if (vendor.SessionsRevokedBefore is not null)
        {
            await using VendorDbContext db = infrastructure.OpenDatabase(vendor);
            await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
            DateTimeOffset cutoff = vendor.SessionsRevokedBefore.Value;
            await db.AgentCredentials.Where(x => x.CreatedAt <= cutoff && x.RevokedAt
== null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, cutoff),
ct);
            await db.AgentEnrollments.Where(x => x.CreatedAt <= cutoff && x.UsedAt ==
null && x.RevokedAt == null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, cutoff),
ct);
            await db.AuthenticationSessions.Where(x => x.CreatedAt <= cutoff &&
x.RevokedAt == null).ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, cutoff), ct);
            await db.PasswordResets.Where(x => x.CreatedAt <= cutoff && x.UsedAt ==
null && x.RevokedAt == null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt,
cutoff).SetProperty(x => x.ProtectedToken, (string?)null), ct);
            await db.AccountActivations.Where(x => x.CreatedAt <= cutoff && x.UsedAt ==
null && x.RevokedAt == null)
                .ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt,
cutoff).SetProperty(x => x.ProtectedToken, (string?)null), ct);
            audit.Record(db, actorId, "VendorCredentialsRevoked", vendorId, entityType:
"SoftwareVendor", actorType: "PlatformAdmin");
            await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
        }
        if (active)
        {
            if (vendor.Database?.Status != "Ready" || vendor.FileArchive?.Status !=
"Ready" || vendor.Provisioning.FailureCode is not null) { return 409; }
            vendor.Status = "Active"; vendor.DeactivatedAt = null;

```

```

        PlatformAudit(vendorId, actorId, "VendorReactivated", changed);
        await platform.SaveChangesAsync(ct);
    }
    return 204;
}

/// <summary>
/// Adds a platform audit entry for a vendor lifecycle change.
/// </summary>
private void PlatformAudit(Guid vendorId, Guid actorId, string action, bool changed) =>
platform.PlatformAuditLogs.Add(new PlatformAuditLog
{
    PlatformAdminId = actorId,
    VendorId = vendorId,
    Action = action,
    EntityType = "SoftwareVendor",
    EntityId = vendorId.ToString(),
    Details = changed ? "{\\"outcome\\":\\"Succeeded\\"}" : "{\\"outcome\\":\\"Unchanged\\"}",
    CreatedAt = clock.GetUtcNow()
});
}

```

VendorProvisioningService.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Platform;
using BridgeControl.Mappings.Shared;
using BridgeControl.Models.Platform;
using BridgeControl.Repositories.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Provisions a vendor's database and archive on demand, retaining completed steps after
failures.
/// </summary>
public class VendorProvisioningService(PlatformDbContext platform, IVendorInfrastructure
infrastructure,
    VendorRoleSeeder roles, IVendorRepository vendors, TimeProvider clock)
{
    /// <summary>
    /// Provisions a vendor identity, database, archive, and initial administrator invitation.
    /// </summary>
    public async Task<VendorDto> CreateAsync(CreateVendorRequest request, Guid
administratorId, CancellationToken cancellationToken)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, cancellationToken);
        SoftwareVendor? vendor = await platform.SoftwareVendors.Include(vendor =>
vendor.Database)
            .Include(vendor => vendor.FileArchive).Include(vendor =>
vendor.Provisioning)
            .SingleOrDefaultAsync(vendor => vendor.Name == request.Name,
cancellationToken);
        if (vendor is not null && (vendor.Status is not ("Provisioning" or "Failed" or
"AwaitingAdministrator") || vendor.Provisioning is null))
        {

```

```

        throw new ApiConflictException("vendor-name-in-use", "A vendor with that
name already exists.");
    }
    if (vendor is null)
    {
        Guid id = Guid.NewGuid();
        DateTimeOffset now = clock.GetUtcNow();
        vendor = new SoftwareVendor
        {
            Id = id,
            Status = "Provisioning",
            CreatedAt = now,
            Database = new VendorDatabase { Id = Guid.NewGuid(), VendorId = id,
DatabaseName = VendorInfrastructure.DatabaseName(id), ConnectionSecretRef =
VendorInfrastructure.ConnectionReference, Status = "Pending", CreatedAt = now },
            FileArchive = new VendorFileArchive { Id = Guid.NewGuid(), VendorId
= id, StorageIdentifier = id.ToString("N"), AccessSecretRef =
VendorInfrastructure.ArchiveReference, Status = "Pending", CreatedAt = now },
            Provisioning = new VendorProvisioning { VendorId = id, UpdatedAt =
now }

        };
        vendor.MapDto(request);
        platform.SoftwareVendors.Add(vendor);
        Audit(vendor.Id, administratorId, "VendorProvisioningStarted");
        await platform.SaveChangesAsync(cancellationToken);
    }
    VendorProvisioning progress = vendor.Provisioning!;
    if (vendor.Status == "AwaitingAdministrator" && progress.FailureCode is null)
    {
        return (await vendors.GetAsync(vendor.Id, cancellationToken))!;
    }
    string stage = "database";
    try
    {
        vendor.Status = "Provisioning";
        progress.FailureCode = null;
        await using VendorDbContext database = infrastructure.OpenDatabase(vendor);
        await database.Database.MigrateAsync(cancellationToken);
        vendor.Database!.Status = "Ready";
        progress.DatabaseReadyAt ??= clock.GetUtcNow();
        progress.SchemaVersion = (await
database.Database.GetAppliedMigrationsAsync(cancellationToken)).Last();
        await SaveProgressAsync(progress, cancellationToken);

        stage = "archive";
        infrastructure.EnsureArchive(vendor);
        vendor.FileArchive!.Status = "Ready";
        progress.ArchiveReadyAt ??= clock.GetUtcNow();
        await SaveProgressAsync(progress, cancellationToken);

        stage = "roles";
        await roles.SeedAsync(database, cancellationToken);
        progress.SeededAt ??= clock.GetUtcNow();
        vendor.Status = "AwaitingAdministrator";
        Audit(vendor.Id, administratorId, "VendorInfrastructureReady");
        await SaveProgressAsync(progress, cancellationToken);
    }
    catch
    {
        // Store only a fixed stage code. Never persist an exception containing
connection details.

        platform.ChangeTracker.Clear();
    }
}

```

```

        SoftwareVendor failed = await platform.SoftwareVendors.Include(v =>
vendor.Provisioning)
            .SingleAsync(v => v.Id == vendor.Id, CancellationToken.None);
        failed.Status = "Failed";
        failed.Provisioning!.FailureCode = stage + "-failed";
        Audit(vendor.Id, administratorId, "VendorProvisioningFailed");
        await SaveProgressAsync(failed.Provisioning, CancellationToken.None);
        throw;
    }
    return (await vendors.GetAsync(vendor.Id, cancellationToken))!;
}

/// <summary>
/// Persists resumable vendor provisioning progress.
/// </summary>
private Task SaveProgressAsync(VendorProvisioning progress, CancellationToken
cancellationToken)
{
    progress.UpdatedAt = clock.UtcNow();
    return platform.SaveChangesAsync(cancellationToken);
}

/// <summary>
/// Adds a platform audit entry for vendor provisioning.
/// </summary>
private void Audit(Guid vendorId, Guid administratorId, string action) =>
platform.PlatformAuditLogs.Add(new PlatformAuditLog
{
    PlatformAdminId = administratorId,
    VendorId = vendorId,
    Action = action,
    EntityType = nameof(SoftwareVendor),
    EntityId = vendorId.ToString(),
    Details = "{}",
    CreatedAt = clock.UtcNow()
});
}

```

VendorServiceRegistration.cs

```

using System.Threading.RateLimiting;
using BridgeControl.Options;
using BridgeControl.Repositories.Platform;
using BridgeControl.Services.Vendor;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Platform;

/// <summary>
/// Registers vendor provisioning and private invitation delivery without opening any vendor
database at startup.
/// </summary>
public static class VendorServiceRegistration
{
    /// <summary>
    /// Registers vendor provisioning and infrastructure services.
    /// </summary>
    public static IServiceCollection AddVendorProvisioning(this IServiceCollection services,
IConfiguration configuration)
    {

```



```

services.AddOptions<VendorProvisioningOptions>().Bind(configuration.GetSection(VendorProvisioning
Options.SectionName))
    .ValidateDataAnnotations().Validate(options =>
Path.IsPathFullyQualified(options.StorageRoot)
    && Path.IsPathFullyQualified(options.InvitationDirectory), "Vendor
storage and invitation paths must be absolute.").ValidateOnStart();
    services.AddDataProtection();
    services.AddSingleton<IVendorInfrastructure, VendorInfrastructure>();
    services.AddSingleton<VendorRoleSeeder>();
    services.AddSingleton<IInvitationDelivery, LocalInvitationDelivery>();
    services.AddScoped<ITenantResolver, TenantResolver>();
    services.AddScoped<IVendorRepository, VendorRepository>();
    services.AddScoped<VendorProvisioningService>();
    services.AddScoped<VendorInvitationService>();
    services.AddScoped<AccountActivationService>();
    services.AddRateLimiter(options =>
    {
        foreach (string policy in new[] { "AccountActivation", "VendorInvitation"
    })
        {
            options.AddPolicy(policy, context =>
RateLimitPartition.GetFixedWindowLimiter(
                $"{policy}:{context.Connection.RemoteIpAddress}", _ => new
FixedWindowRateLimiterOptions
                {
                    PermitLimit =
context.RequestServices.GetRequiredService<Options<VendorProvisioningOptions>>().Value.Activatio
nRequestsPerMinute,
                    Window = TimeSpan.FromMinutes(1),
                    QueueLimit = 0,
                    AutoReplenishment = true
                });
        }
    });
    return services;
}
}

```

Services

ITenantResolver.cs

```

using System.Security.Claims;

namespace BridgeControl.Services;

/// <summary>
/// Resolves active vendors before login or from an already authenticated vendor-user identity.
/// </summary>
public interface ITenantResolver
{
    // The locator is untrusted; only active server-owned metadata supplies infrastructure.
    /// <summary>
    /// Resolves an active vendor context for an agent identity.
    /// </summary>
}

```

```

        Task<TenantContext?> ResolveForAgentAsync(Guid vendorId, CancellationToken
cancellationToken = default);

        /// <summary>
        /// Resolves an active vendor context by its login name.
        /// </summary>
        Task<TenantContext?> ResolveForLoginAsync(string vendorName, CancellationToken
cancellationToken = default);

        /// <summary>
        /// Resolves the vendor context identified by the authenticated principal.
        /// </summary>
        Task<TenantContext?> ResolveAuthenticatedAsync(ClaimsPrincipal user, CancellationToken
cancellationToken = default);
    }

```

Services/Shared

AccountValidation.cs

```

using System.ComponentModel.DataAnnotations;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Validates new human account credentials without changing the entered password.
/// </summary>
public static class AccountValidation
{
    public const int MinimumPasswordLength = 10;
    public const int MaximumPasswordLength = 128;

    /// <summary>
    /// Trims and normalizes an email address for account lookup.
    /// </summary>
    public static string NormalizeEmail(string email) => email.Trim().ToLowerInvariant();

    /// <summary>
    /// Checks the email and password requirements for a new account.
    /// </summary>
    public static string? ValidateNewAccount(string email, string password)
    {
        if (string.IsNullOrEmpty(email) || email.Length > 254 || !new
EmailAddressAttribute().IsValid(email))
        {
            return "Enter a valid email address of at most 254 characters.";
        }

        if (string.IsNullOrEmpty(password) || password.Length < MinimumPasswordLength
|| password.Length > MaximumPasswordLength)
        {
            return $"Choose a password between {MinimumPasswordLength} and
{MaximumPasswordLength} characters.";
        }

        return null;
    }
}

```

```
}  
}
```

AuthenticationSessionService.cs

```
using BridgeControl.Models.Shared;  
using Microsoft.EntityFrameworkCore;  
using Microsoft.EntityFrameworkCore.Storage;  
  
namespace BridgeControl.Services.Shared;  
  
/// <summary>  
/// Creates, rotates and revokes human sessions in the database selected by the calling  
authentication flow.  
/// </summary>  
public class AuthenticationSessionService(RefreshCredentialService credentials, TimeProvider  
clock)  
{  
    /// <summary>  
    /// Creates an authentication session and its initial refresh credential.  
    /// </summary>  
    public IssuedSession Create(Guid accountId, TimeSpan sessionLifetime, TimeSpan  
refreshLifetime)  
    {  
        DateTimeOffset now = clock.GetUtcNow();  
        AuthenticationSession session = new()  
        {  
            Id = Guid.NewGuid(),  
            AccountId = accountId,  
            CreatedAt = now,  
            ExpiresAt = now.Add(sessionLifetime)  
        };  
        return Issue(session, now, refreshLifetime);  
    }  
  
    /// <summary>  
    /// Atomically replaces a refresh credential and detects reuse of consumed credentials.  
    /// </summary>  
    public async Task<IssuedSession?> RotateAsync(DbContext database, string? credential,  
        TimeSpan refreshLifetime, Func<Guid, CancellationToken, Task<bool>>  
accountIsActive,  
        Action<AuthenticationSession, string> recordEvent, CancellationToken  
cancellationToken)  
    {  
        if (!credentials.TryHash(credential, out byte[] hash))  
        {  
            return null;  
        }  
  
        AuthenticationSession? session = null;  
        try  
        {  
            await using IDbContextTransaction transaction = await  
database.Database.BeginTransactionAsync(cancellationToken);  
            RefreshToken? token = await database.Set<RefreshToken>()  
                .Include(entity => entity.AuthenticationSession)  
                .SingleOrDefaultAsync(entity => entity.TokenHash == hash,  
cancellationToken);  
  
            DateTimeOffset now = clock.GetUtcNow();  
            session = token?.AuthenticationSession;
```

```

        // JWT expirations use whole seconds; do not rotate into an already-expired
access token.
        if (session is null || session.RevokedAt is not null ||
session.ExpiresAt.ToUnixTimeSeconds() <= now.ToUnixTimeSeconds())
        {
            return null;
        }

        if (!await accountIsActive(session.AccountId, cancellationToken))
        {
            await database.Set<AuthenticationSession>()
                .Where(entity => entity.AccountId == session.AccountId &&
entity.RevokedAt == null)
                .ExecuteUpdateAsync(setters => setters.SetProperty(entity =>
entity.RevokedAt, now), cancellationToken);
            recordEvent(session, "AccountSessionsRevoked");
            await database.SaveChangesAsync(cancellationToken);
            await transaction.CommitAsync(cancellationToken);
            return null;
        }

        if (token!.UsedAt is not null)
        {
            session.RevokedAt = now;
            recordEvent(session, "RefreshCredentialReused");
            await database.SaveChangesAsync(cancellationToken);
            await transaction.CommitAsync(cancellationToken);
            return null;
        }

        if (token.ExpiresAt <= now)
        {
            return null;
        }

        token.UsedAt = now;
        session.LastRefreshedAt = now;
        // Always update this row so concurrent refresh/logout operations conflict.
        database.Entry(session).Property(entity =>
entity.LastRefreshedAt).IsModified = true;
        IssuedSession issued = Issue(session, now, refreshLifetime);
        // New rows have explicit IDs; mark the replacement as Added before change
detection.

        database.Set<RefreshToken>().Add(session.RefreshTokens.Last());
        recordEvent(session, "SessionRefreshed");
        await database.SaveChangesAsync(cancellationToken);
        await transaction.CommitAsync(cancellationToken);
        return issued;
    }
    catch (DbUpdateConcurrencyException) when (session is not null)
    {
        // A competing refresh or revocation won. Invalidate the entire token
family.

        database.ChangeTracker.Clear();
        await RevokeAsync(database, session.Id, session.AccountId, recordEvent,
            "RefreshCredentialReused", cancellationToken);
        return null;
    }
}

/// <summary>
/// Revokes a session and its remaining refresh credentials.

```

```

    /// </summary>
    public async Task RevokeAsync(DbContext database, Guid sessionId, Guid accountId,
        Action<AuthenticationSession, string> recordEvent, string action,
        CancellationToken cancellationToken)
    {
        await using IDbContextTransaction transaction = await
            database.Database.BeginTransactionAsync(cancellationToken);
        DateTimeOffset now = clock.GetUtcNow();
        int changed = await database.Set<AuthenticationSession>()
            .Where(entity => entity.Id == sessionId && entity.AccountId == accountId &&
                entity.RevokedAt == null)
            .ExecuteUpdateAsync(setters => setters.SetProperty(entity =>
                entity.RevokedAt, now), cancellationToken);
        if (changed > 0)
        {
            recordEvent(new AuthenticationSession { Id = sessionId, AccountId =
                accountId }, action);
            await database.SaveChangesAsync(cancellationToken);
        }
        await transaction.CommitAsync(cancellationToken);
    }

    /// <summary>
    /// Creates a refresh credential and records its hash for the session.
    /// </summary>
    private IssuedSession Issue(AuthenticationSession session, DateTimeOffset now, TimeSpan
        refreshLifetime)
    {
        string credential = credentials.Create();
        credentials.TryHash(credential, out byte[] hash);
        DateTimeOffset expiry = now.Add(refreshLifetime);
        if (expiry > session.ExpiresAt)
        {
            expiry = session.ExpiresAt;
        }
        session.RefreshTokens.Add(new RefreshToken
        {
            Id = Guid.NewGuid(),
            AuthenticationSessionId = session.Id,
            AuthenticationSession = session,
            TokenHash = hash,
            CreatedAt = now,
            ExpiresAt = expiry
        });
        return new IssuedSession(session, credential, expiry);
    }
}

```

InteractiveCredentials.cs

```

using System.Text;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Reads an operator-entered password without echoing or changing its characters.
/// </summary>
internal static class InteractiveCredentials
{
    /// <summary>

```

```

    /// Reads a password from the console without echoing its characters.
    /// </summary>
    public static string ReadPassword()
    {
        StringBuilder input = new();
        while (true)
        {
            ConsoleKeyInfo key = Console.ReadKey(intercept: true);
            if (key.Key == ConsoleKey.Enter)
            {
                Console.WriteLine();
                return input.ToString();
            }
            if (key.Key == ConsoleKey.Backspace && input.Length > 0)
            {
                input.Length--;
            }
            else if (!char.IsControl(key.KeyChar))
            {
                if (input.Length == AccountValidation.MaximumPasswordLength)
                {
                    throw new ArgumentException($"The password must be at most {AccountValidation.MaximumPasswordLength} characters. No account was changed.");
                }
                input.Append(key.KeyChar);
            }
        }
    }
}

```

IssuedSession.cs

```

using BridgeControl.Models.Shared;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Carries a session and its new refresh credential internally until the response is sent.
/// </summary>
public sealed class IssuedSession(AuthenticationSession session, string refreshCredential,
    DateTimeOffset refreshExpiresAt)
{
    public AuthenticationSession Session { get; } = session;
    public string RefreshCredential { get; } = refreshCredential;
    public DateTimeOffset RefreshExpiresAt { get; } = refreshExpiresAt;
}

```

PagedResponseExtensions.cs

```

using BridgeControl.DTOs.Shared;

namespace BridgeControl.Services.Shared;

/// <summary>Maps materialized pages while preserving their pagination metadata.</summary>
public static class PagedResponseExtensions
{
    {

```

```

        /// <summary>Maps items sequentially so callbacks can safely share a database
context.</summary>
        public static async Task<PagedResponse<TResult>> MapAsync<TSource, TResult>(this
PagedResponse<TSource> page,
            Func<TSource, CancellationToken, Task<TResult>> map, CancellationToken
cancellationToken = default)
        {
            ArgumentNullException.ThrowIfNull(page);
            ArgumentNullException.ThrowIfNull(map);
            cancellationToken.ThrowIfCancellationRequested();
            List<TResult> items = new(page.Items.Count);
            foreach (TSource item in page.Items)
            {
                cancellationToken.ThrowIfCancellationRequested();
                items.Add(await map(item, cancellationToken));
            }
            return new() { Items = items, Page = page.Page, PageSize = page.PageSize,
TotalCount = page.TotalCount };
        }
    }
}

```

PaginationExtensions.cs

```

using System.ComponentModel.DataAnnotations;
using System.Linq.Expressions;
using BridgeControl.DTOs.Shared;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Counts and pages an already authorized query using an explicit stable unique ordering key.
/// </summary>
public static class PaginationExtensions
{
    /// <summary>
    /// Executes a bounded query page and returns its pagination metadata.
    /// </summary>
    public static Task<PagedResponse<T>> ToPageAsync<T, TKey>(this IQueryable<T> scopedQuery,
        PaginationQuery pagination, Expression<Func<T, TKey>> uniqueOrderKey,
        CancellationToken cancellationToken = default,
        bool descending = false)
    => scopedQuery.ToPageAsync(pagination, uniqueOrderKey, item => item,
        cancellationToken, descending);

    /// <summary>
    /// Selects response columns after applying the same scope, bounds and ordering used for
the page count.
    /// </summary>
    public static async Task<PagedResponse<TResponse>> ToPageAsync<T, TKey, TResponse>(this
IQueryable<T> scopedQuery,
        PaginationQuery pagination, Expression<Func<T, TKey>> uniqueOrderKey,
        Expression<Func<T, TResponse>> projection, CancellationToken cancellationToken =
default, bool descending = false)
    {
        Validator.ValidateObject(pagination, new ValidationContext(pagination),
validateAllProperties: true);
        IOOrderedQueryable<T> ordered = descending ?
scopedQuery.OrderByDescending(uniqueOrderKey) : scopedQuery.OrderBy(uniqueOrderKey);
        return new PagedResponse<TResponse>

```

```

        {
            Page = pagination.Page,
            PageSize = pagination.PageSize,
            TotalCount = await scopedQuery.CountAsync(cancellationToken),
            Items = await ordered.Skip(pagination.Offset)

        }.Take(pagination.PageSize).Select(projection).ToListAsync(cancellationToken)
    };
}
}

```

PasswordService.cs

```

using System.Security.Cryptography;
using Microsoft.AspNetCore.Identity;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Hashes and verifies passwords for either account type, including a dummy check for unknown
/// accounts.
/// </summary>
public class PasswordService<TAccount>(IPasswordHasher<TAccount> hasher) where TAccount : class,
new()
{
    private readonly TAccount dummyAccount = new();
    private readonly string dummyHash = hasher.HashPassword(new TAccount(),
        Convert.ToBase64String(RandomNumberGenerator.GetBytes(32)));

    /// <summary>
    /// Hashes a password using the configured password hasher.
    /// </summary>
    public string Hash(TAccount account, string password) => hasher.HashPassword(account,
password);

    /// <summary>
    /// Verifies a password against its stored hash.
    /// </summary>
    public PasswordVerificationResult Verify(TAccount? account, string? storedHash, string
password)
    {
        if (account is null || string.IsNullOrEmpty(storedHash))
        {
            hasher.VerifyHashedPassword(dummyAccount, dummyHash, password);
            return PasswordVerificationResult.Failed;
        }

        try
        {
            return hasher.VerifyHashedPassword(account, storedHash, password);
        }
        catch (FormatException)
        {
            return PasswordVerificationResult.Failed;
        }
    }
}

```


PrivateDirectory.cs

```
using System.Security.AccessControl;
using System.Security.Principal;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Creates private service-owned directories and refuses paths through symbolic links or
junctions.
/// </summary>
public static class PrivateDirectory
{
    /// <summary>
    /// Creates a directory and restricts its filesystem permissions.
    /// </summary>
    public static string Ensure(string path, string? additionalServiceSid = null)
    {
        if (!Path.IsPathFullyQualified(path))
        {
            throw new InvalidOperationException("Private storage must use an absolute
path.");
        }
        string fullPath = Path.GetFullPath(path);
        string[] protectedRoots = [Path.GetPathRoot(fullPath)!,
Environment.GetFolderPath(Environment.SpecialFolder.UserProfile),
Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)];
        if (protectedRoots.Any(root =>
string.Equals(Path.TrimEndingDirectorySeparator(root),
Path.TrimEndingDirectorySeparator(fullPath), StringComparison.OrdinalIgnoreCase)))
        {
            throw new InvalidOperationException("Private storage must use its own
dedicated directory.");
        }
        for (DirectoryInfo? current = new(fullPath); current is not null; current =
current.Parent)
        {
            if (current.Exists &&
current.Attributes.HasFlag(FileAttributes.ReparsePoint))
            {
                throw new InvalidOperationException("Private storage cannot use a
symbolic link or junction.");
            }
        }
        if (OperatingSystem.IsWindows())
        {
            SecurityIdentifier serviceUser = WindowsIdentity.GetCurrent().User!;
            SecurityIdentifier system = new(WellKnownSidType.LocalSystemSid, null);
            SecurityIdentifier? additional = additionalServiceSid is null ? null : new
SecurityIdentifier(additionalServiceSid);
            if (additional is not null &&
(additional.IsWellKnown(WellKnownSidType.WorldSid) ||
additional.IsWellKnown(WellKnownSidType.AuthenticatedUserSid)
|| additional.IsWellKnown(WellKnownSidType.BuiltinUsersSid) ||
additional.IsWellKnown(WellKnownSidType.BuiltinAdministratorsSid)))
            { throw new InvalidOperationException("Use a dedicated SQL Server service
SID, not a shared access group."); }
            DirectorySecurity security = new();
            security.SetAccessRuleProtection(isProtected: true, preserveInheritance:
false);
```

```

        InheritanceFlags inheritance = InheritanceFlags.ContainerInherit |
InheritanceFlags.ObjectInherit;
        security.AddAccessRule(new FileSystemAccessRule(serviceUser,
FileSystemRights.FullControl,
            inheritance, PropagationFlags.None, AccessControlType.Allow));
        security.AddAccessRule(new FileSystemAccessRule(system,
            FileSystemRights.FullControl, inheritance, PropagationFlags.None,
AccessControlType.Allow));
        if (additional is not null) { security.AddAccessRule(new
FileSystemAccessRule(additional, FileSystemRights.Modify, inheritance, PropagationFlags.None,
AccessControlType.Allow)); }
        DirectoryInfo directory = security.CreateDirectory(fullPath);
        // CreateDirectory preserves an existing directory. Validate it rather than
changing an unrelated folder's ACL.
        IEnumerable<FileSystemAccessRule> rules =
directory.GetAccessControl().GetAccessRules(true, true,
typeof(SecurityIdentifier)).Cast<FileSystemAccessRule>();
        foreach (FileSystemAccessRule rule in rules)
        {
            if (rule.AccessControlType == AccessControlType.Allow &&
rule.IdentityReference != serviceUser && rule.IdentityReference != system &&
rule.IdentityReference != additional)
            {
                throw new InvalidOperationException("Private storage must
allow access only to the service account and SYSTEM.");
            }
        }
    }
    else
    {
        Directory.CreateDirectory(fullPath, UnixFileMode.UserRead |
UnixFileMode.UserWrite | UnixFileMode.UserExecute);
        UnixFileMode otherAccess = UnixFileMode.GroupRead | UnixFileMode.GroupWrite
| UnixFileMode.GroupExecute
        | UnixFileMode.OtherRead | UnixFileMode.OtherWrite |
UnixFileMode.OtherExecute;
        if ((File.GetUnixFileMode(fullPath) & otherAccess) != 0)
        {
            throw new InvalidOperationException("Private storage must allow
access only to its owner.");
        }
        return fullPath;
    }
}

```

RefreshCredentialService.cs

```

using System.Security.Cryptography;
using System.Text;
using Microsoft.AspNetCore.WebUtilities;

namespace BridgeControl.Services.Shared;

/// <summary>
/// Generates random refresh credentials and derives the hashes stored in either database.
/// </summary>
public class RefreshCredentialService
{
    /// <summary>

```

```

    /// Creates a random refresh credential and its storage hash.
    /// </summary>
    public string Create() => WebEncoders.Base64UrlEncode(RandomNumberGenerator.GetBytes(32));

    /// <summary>
    /// Validates and hashes a supplied refresh credential without storing its plaintext.
    /// </summary>
    public bool TryHash(string? credential, out byte[] hash)
    {
        hash = [];
        if (credential is null || credential.Length != 43)
        {
            return false;
        }

        try
        {
            byte[] bytes = WebEncoders.Base64UrlDecode(credential);
            if (bytes.Length != 32 || WebEncoders.Base64UrlEncode(bytes) != credential)
            {
                return false;
            }

            hash = SHA256.HashData(Encoding.UTF8.GetBytes(credential));
            return true;
        }
        catch (FormatException)
        {
            return false;
        }
    }
}

```

Services

TenantContext.cs

```

using BridgeControl.Data;

namespace BridgeControl.Services;

/// <summary>
/// Owns the database context for one resolved active vendor; callers dispose it after the
operation.
/// </summary>
public sealed class TenantContext(Guid vendorId, VendorDbContext database, string archivePath) :
    IAsyncDisposable
{
    public Guid VendorId { get; } = vendorId;
    public VendorDbContext Database { get; } = database;
    public string ArchivePath { get; } = archivePath;

    /// <summary>
    /// Asynchronously releases the resources owned by this instance.
    /// </summary>
    public ValueTask DisposeAsync() => Database.DisposeAsync();
}

```

```
}
```

TenantResolver.cs

```
using System.Security.Claims;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services;

/// <summary>
/// Resolves trusted infrastructure only after checking current vendor and provisioning state.
/// </summary>
public class TenantResolver(PlatformDbContext platform, IVendorInfrastructure infrastructure) :
ITenantResolver
{
    /// <summary>
    /// Resolves an active vendor context for an agent identity.
    /// </summary>
    public Task<TenantContext?> ResolveForAgentAsync(Guid vendorId, CancellationToken
cancellationToken = default)
        => ResolveAsync(platform.SoftwareVendors.Where(vendor => vendor.Id == vendorId),
cancellationToken);

    /// <summary>
    /// Resolves an active vendor context by its login name.
    /// </summary>
    public Task<TenantContext?> ResolveForLoginAsync(string vendorName, CancellationToken
cancellationToken = default)
    {
        string name = vendorName.Trim();
        return ResolveAsync(platform.SoftwareVendors.Where(vendor => vendor.Name == name),
cancellationToken);
    }

    /// <summary>
    /// Resolves the vendor context identified by the authenticated principal.
    /// </summary>
    public Task<TenantContext?> ResolveAuthenticatedAsync(ClaimsPrincipal user,
CancellationToken cancellationToken = default)
    {
        if (user.Identity?.IsAuthenticated != true ||
user.FindFirst("identity_type")?.Value != "vendor_user"
        || !Guid.TryParse(user.FindFirst("vendor_id")?.Value, out Guid id) || id ==
Guid.Empty)
        {
            return Task.FromResult<TenantContext?>(null);
        }
        return ResolveAsync(platform.SoftwareVendors.Where(vendor => vendor.Id == id),
cancellationToken);
    }

    /// <summary>
    /// Loads vendor infrastructure and creates the tenant context when the vendor is
available.
    /// </summary>
    private async Task<TenantContext?> ResolveAsync(IQueryable<SoftwareVendor> query,
CancellationToken cancellationToken)
```

```

        {
            SoftwareVendor? vendor = await query.AsNoTracking().Include(vendor =>
vendor.Database)
                .Include(vendor => vendor.FileArchive).Include(vendor =>
vendor.Provisioning).SingleOrDefaultAsync(cancellationToken);
            if (vendor?.Status != "Active" || vendor.Database?.Status != "Ready" ||
vendor.FileArchive?.Status != "Ready"
                || vendor.Provisioning?.InvitationEstablishedAt is null ||
vendor.Provisioning.DatabaseReadyAt is null
                || vendor.Provisioning.ArchiveReadyAt is null ||
vendor.Provisioning.SeededAt is null || vendor.Provisioning.FailureCode is not null)
            {
                return null;
            }
            try
            {
                string archive = infrastructure.GetArchivePath(vendor);
                return new TenantContext(vendor.Id, infrastructure.OpenDatabase(vendor),
archive);
            }
            catch (InvalidOperationException)
            {
                // Invalid internal references are indistinguishable from an unavailable
vendor to callers.
                return null;
            }
        }
    }
}

```

Services/Vendor

AccountActivationService.cs

```

using System.Security.Cryptography;
using BridgeControl.Data;
using BridgeControl.DTOs.Vendor;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Consumes an invitation once and stores the chosen password hash with its audit event
atomically.
/// </summary>
public class AccountActivationService(PlatformDbContext platform, IVendorInfrastructure
infrastructure,
    RefreshCredentialService credentials, PasswordService<UserAccount> passwords, TimeProvider
clock)
{
    /// <summary>

```

```

        /// Consumes a valid activation credential and activates the account with its new
password.
        /// </summary>
        public async Task<bool> ActivateAsync(ActivateAccountRequest request, CancellationToken
cancellationToken)
        {
            string[] parts = request.Token.Split('.');
            if (parts.Length != 3 || !Guid.TryParseExact(parts[0], "N", out Guid vendorId) ||
vendorId == Guid.Empty
                || !Guid.TryParseExact(parts[1], "N", out Guid activationId) ||
activationId == Guid.Empty
                || !credentials.TryHash(parts[2], out byte[] hash))
            {
                return false;
            }
            await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, cancellationToken);
            SoftwareVendor? vendor = await
platform.SoftwareVendors.AsNoTracking().Include(vendor => vendor.Database)
                .Include(vendor => vendor.FileArchive).Include(vendor =>
vendor.Provisioning).SingleOrDefaultAsync(vendor => vendor.Id == vendorId, cancellationToken);
            if (vendor?.Status != "Active" || vendor.Database?.Status != "Ready" ||
vendor.FileArchive?.Status != "Ready"
                || vendor.Provisioning?.InvitationEstablishedAt is null ||
vendor.Provisioning.DatabaseReadyAt is null
                || vendor.Provisioning.ArchiveReadyAt is null ||
vendor.Provisioning.SeededAt is null || vendor.Provisioning.FailureCode is not null)
            {
                return false;
            }
            await using VendorDbContext database = infrastructure.OpenDatabase(vendor);
            await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(cancellationToken);
            AccountActivation? activation = await database.AccountActivations.Include(token =>
token.User).ThenInclude(user => user.Customer)
                .SingleOrDefaultAsync(token => token.Id == activationId,
cancellationToken);
            DateTimeOffset now = clock.UtcNow();
            if (activation is null || activation.UsedAt is not null || activation.RevokedAt is
not null || activation.DeliveredAt is null
                || activation.ExpiresAt <= now || activation.User.Status != "Invited"
                || (activation.User.CustomerId is not null &&
activation.User.Customer?.Status != "Active")
                || !CryptographicOperations.FixedTimeEquals(activation.TokenHash, hash)
                || AccountValidation.ValidateNewAccount(activation.User.Email,
request.Password) is not null)
            {
                return false;
            }
            activation.User.PasswordHash = passwords.Hash(activation.User, request.Password);
            activation.User.Status = "Active";
            activation.UsedAt = now;
            activation.ProtectedToken = null;
            await database.AuthenticationSessions.Where(session => session.AccountId ==
activation.UserId && session.RevokedAt == null)
                .ExecuteUpdateAsync(setters => setters.SetProperty(session =>
session.RevokedAt, now), cancellationToken);
            database.VendorAuditLogs.Add(new VendorAuditLog
            {
                ActorId = activation.UserId,
                ActorType = "User",
                EntityId = activation.UserId,

```

```

        Action = "AccountActivated",
        CreatedAt = now
    });
    await database.SaveChangesAsync(cancellationToken);
    await transaction.CommitAsync(cancellationToken);
    return true;
}
}

```

AccountInvitationService.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>Shared invitation verification and retryable delivery for bootstrap and user
administration.</summary>
public class AccountInvitationService(IInvitationDelivery delivery, RefreshCredentialService
credentials,
    IDataProtectionProvider protection, IOptions<VendorProvisioningOptions> options,
    VendorSecurityAudit audit, TimeProvider clock)
{
    // Caller holds the vendor write lock and commits user/role/audit changes with this
    operation.
    /// <summary>
    /// Creates or reuses the pending account activation invitation.
    /// </summary>
    public async Task<AccountActivation> EnsureAsync(VendorDbContext db, Guid vendorId,
    UserAccount user, Guid actorId,
        string action, string actorType, CancellationToken ct)
    {
        AccountActivation? activation = await db.AccountActivations.SingleOrDefaultAsync(x
=> x.UserId == user.Id && x.UsedAt == null && x.RevokedAt == null, ct);
        if (activation is not null && activation.ExpiresAt <= clock.GetUtcNow())
        { activation.RevokedAt = clock.GetUtcNow(); activation.ProtectedToken = null;
        await db.SaveChangesAsync(ct); activation = null; }
        if (activation is null)
        {
            string secret = credentials.Create(); credentials.TryHash(secret, out
byte[] hash);

            Guid id = Guid.NewGuid();
            activation = new AccountActivation
            {
                Id = id,
                User = user,
                UserId = user.Id,
                TokenHash = hash,
                CreatedAt = clock.GetUtcNow(),
                ExpiresAt =
clock.GetUtcNow().AddHours(options.Value.ActivationHours),
                ProtectedToken =
Protector(vendorId).Protect($"{vendorId:N}.{id:N}.{secret}")
            };
            db.AccountActivations.Add(activation);

```

```

        audit.Record(db, actorId, action, user.Id, actorType: actorType);
    }
    return activation;
}

/// <summary>
/// Delivers the account activation credential to the intended recipient.
/// </summary>
public async Task DeliverAsync(VendorDbContext db, Guid vendorId, UserAccount user,
AccountActivation activation, CancellationToken ct)
{
    if (activation.DeliveredAt is not null) { return; }
    await delivery.DeliverAsync(activation.Id, user.Email,
Protector(vendorId).Unprotect(activation.ProtectedToken!), activation.ExpiresAt, ct);
    activation.DeliveredAt = clock.UtcNow(); activation.ProtectedToken = null;
    await db.SaveChangesAsync(ct);
}

/// <summary>
/// Creates the data protector used for account invitation credentials.
/// </summary>
private IDataProtector Protector(Guid vendorId) =>
protection.CreateProtector("BridgeControl.ActivationDelivery.v1", vendorId.ToString("N"));
}

```

AuditHistoryOperator.cs

```

using System.ComponentModel.DataAnnotations;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Shared;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Shared;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents audit history query data.
/// </summary>
public sealed record AuditHistoryQuery(DateTimeOffset From, DateTimeOffset To, Guid? ActorId =
null, Guid? ResourceId = null, string? Action = null, string? Outcome = null);

/// <summary>
/// Represents audit history row data.
/// </summary>
public sealed record AuditHistoryRow(Guid Id, DateTimeOffset CreatedAt, Guid ActorId, string
ActorType, string EntityType, string EntityId, string Action, string Outcome, string Context);

/// <summary>Only reachable through trusted host maintenance. No HTTP controller or Agent DTO
extension.</summary>
public sealed class AuditHistoryOperator(PlatformDbContext platform, HistoryRetentionService
storage)
{
    /// <summary>
    /// Checks audit search filters and pagination limits.
    /// </summary>
    private static void Validate(AuditHistoryQuery query, PaginationQuery page)
    {

```



```

        if (query.From == default || query.To <= query.From || query.To - query.From >
            TimeSpan.FromDays(31)
            || query.Action?.Length > 100 || query.Outcome?.Length > 32) { throw new
            ArgumentException("Use a time range of at most 31 days and bounded filters."); }
        Validator.ValidateObject(page, new ValidationContext(page), true);
    }

    /// <summary>
    /// Reads a page of vendor audit history for an authorized operator.
    /// </summary>
    public async Task<PagedResponse<AuditHistoryRow>> VendorAsync(Guid vendorId, Guid
viewerUserId, AuditHistoryQuery filter,
        PaginationQuery page, Guid? customerId = null, Guid? installationId = null,
        CancellationToken ct = default)
    {
        Validate(filter, page);
        await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using VendorDbContext db = await storage.OpenVendorAsync(vendorId, ct);
        TenantAccess access = await TenantAccess.LoadAsync(db, viewerUserId, ct) ?? throw
new ApiAccessException(403);
        if (!access.HasPermission("logs.read")) { throw new ApiAccessException(403); }
        IQueryable<Installation> installations = access.Installations(db, "logs.read");
        IQueryable<Customer> customers = access.Customers(db, "logs.read");
        if (customerId is Guid c)
        {
            if (!await customers.AnyAsync(x => x.Id == c, ct)) { throw new
ApiAccessException(404); }
            installations = installations.Where(x => x.CustomerId == c); customers =
customers.Where(x => x.Id == c);
        }
        if (installationId is Guid i)
        {
            if (!await installations.AnyAsync(x => x.Id == i, ct)) { throw new
ApiAccessException(404); }
            installations = installations.Where(x => x.Id == i);
        }
        bool wholeVendor = access.User.CustomerId == null &&
access.Roles.Contains("VendorAdministrator") && customerId == null && installationId == null;
        bool wholeCustomer = installationId == null && (access.User.CustomerId == null ||
access.Roles.Contains("CustomerAdministrator"));
        IQueryable<VendorAuditLog> query = db.VendorAuditLogs.Where(x => wholeVendor
            || (x.EntityType == "Installation" && installations.Any(i => i.Id ==
x.EntityId))
            || (x.EntityType == "Deployment" && db.Deployments.Any(d => d.Id ==
x.EntityId && installations.Any(i => i.Id == d.InstallationId)))
            || (x.EntityType == "ConfigurationProfile" &&
db.ConfigurationProfiles.Any(p => p.Id == x.EntityId && (installations.Any(i => i.Id ==
p.InstallationId) || (wholeCustomer && customers.Any(c => c.Id == p.CustomerId))))))
            || (wholeCustomer && ((x.EntityType == "Customer" && customers.Any(c =>
c.Id == x.EntityId))
                || ((x.EntityType == "UserAccount" || x.EntityType == "UserRole")
&& db.UserAccounts.Any(u => u.Id == x.EntityId && customers.Any(c => c.Id == u.CustomerId)))
                || (x.EntityType == "Agent" && db.Agents.Any(a => a.Id ==
x.EntityId && customers.Any(c => c.Id == a.CustomerId)))
                || (x.EntityType == "AgentEnrollment" && db.AgentEnrollments.Any(a
=> a.Id == x.EntityId && customers.Any(c => c.Id == a.CustomerId)))
                || (x.EntityType == "License" && db.Licenses.Any(l => l.Id ==
x.EntityId && customers.Any(c => c.Id == l.CustomerId)))));
        query = query.Where(x => x.CreatedAt >= filter.From && x.CreatedAt < filter.To);
        if (filter.ActorId != null) { query = query.Where(x => x.ActorId ==
filter.ActorId); }
    }

```

```

        if (filter.ResourceId != null) { query = query.Where(x => x.EntityId ==
filter.ResourceId); }
        if (filter.Action != null) { query = query.Where(x => x.Action == filter.Action);
}
        if (filter.Outcome != null) { query = query.Where(x => x.Outcome ==
filter.Outcome); }
        PagedResponse<AuditHistoryRow> result = await new Repository<VendorAuditLog,
AuditHistoryRow>(query, x => new(x.Id, x.CreatedAt, x.ActorId, x.ActorType, x.EntityType,
x.EntityId.ToString(), x.Action, x.Outcome, x.Context)).PageAsync(page, x => x.Id, ct);
        return Sanitize(result);
    }

    /// <summary>
    /// Reads a page of platform audit history for an authorized administrator.
    /// </summary>
    public async Task<PagedResponse<AuditHistoryRow>> PlatformAsync(Guid viewerAdminId,
AuditHistoryQuery filter, PaginationQuery page, Guid? vendorId = null, CancellationToken ct =
default)
    {
        Validate(filter, page);
        await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
        if (!await platform.PlatformAdmins.AnyAsync(x => x.Id == viewerAdminId && x.Status
== "Active", ct)) { throw new ApiAccessException(403); }
        IQueryable<PlatformAuditLog> query = platform.PlatformAuditLogs.Where(x =>
x.CreatedAt >= filter.From && x.CreatedAt < filter.To);
        if (vendorId != null) { query = query.Where(x => x.VendorId == vendorId); }
        if (filter.ActorId != null) { query = query.Where(x => x.PlatformAdminId ==
filter.ActorId); }
        if (filter.ResourceId != null) { string resource = filter.ResourceId.ToString();
query = query.Where(x => x.EntityId == resource); }
        if (filter.Action != null) { query = query.Where(x => x.Action == filter.Action);
}

        // Platform writers use only these two outcome documents; all other legacy entries
        imply success.
        if (filter.Outcome == "Unchanged") { query = query.Where(x => x.Details ==
"{\"outcome\":\"Unchanged\"}"); }
        else if (filter.Outcome == "Succeeded") { query = query.Where(x => x.Details !=
"{\"outcome\":\"Unchanged\"}"); }
        else if (filter.Outcome != null) { query = query.Where(x => false); }
        PagedResponse<AuditHistoryRow> result = await new
Repository<BridgeControl.Models.Platform.PlatformAuditLog, AuditHistoryRow>(query,
x => new(x.Id, x.CreatedAt, x.PlatformAdminId, "PlatformAdministrator",
x.EntityType, x.EntityId, x.Action,
x.Details == "{\"outcome\":\"Unchanged\"}" ? "Unchanged" :
"Succeeded", x.Details)).PageAsync(page, x => x.Id, ct);
        return Sanitize(result);
    }

    /// <summary>
    /// Redacts sensitive context from audit rows before returning a page.
    /// </summary>
    private static PagedResponse<AuditHistoryRow> Sanitize(PagedResponse<AuditHistoryRow>
page)
    {
        page.Items = page.Items.Select(x => x with { Context = SafeContext(x.Context)
}).ToArray(); return page;
    }

    /// <summary>
    /// Returns bounded audit context with sensitive values removed.
    /// </summary>

```

```

public static string SafeContext(string context)
{
    Dictionary<string, object?> safe = [];
    try
    {
        using JsonDocument doc = JsonDocument.Parse(context);
        if (doc.RootElement.ValueKind != JsonValueKind.Object) { return "{}"; }
        foreach (JsonProperty p in doc.RootElement.EnumerateObject())
        {
            if (p.Name is "resourceId" or "reviewId" or "sourceRevisionId" or
"resultRevisionId" or "ruleId"
                && p.Value.ValueKind == JsonValueKind.String &&
p.Value.TryGetGuid(out Guid id)) { safe[p.Name] = id; }
            else if (p.Name == "changeIndex" && p.Value.ValueKind ==
JsonValueKind.Number && p.Value.TryGetInt32(out int index)) { safe[p.Name] = index; }
            else if (p.Name == "previousValueHash" && p.Value.ValueKind ==
JsonValueKind.String && p.Value.GetString() is string hash && hash.Length == 64 &&
hash.All(Uri.IsHexDigit)) { safe[p.Name] = hash; }
            else if (p.Name == "layer" && p.Value.ValueKind ==
JsonValueKind.String && p.Value.GetString() is "Product" or "Customer" or "Installation") {
safe[p.Name] = p.Value.GetString(); }
            else if (p.Name == "outcome" && p.Value.ValueKind ==
JsonValueKind.String && p.Value.GetString() is "Succeeded" or "Unchanged") { safe[p.Name] =
p.Value.GetString(); }
            else if (p.Name == "mechanism" && p.Value.ValueKind ==
JsonValueKind.String && p.Value.GetString() == "offline-operator") { safe[p.Name] =
"offline-operator"; }
            else if (p.Name is "operatorAccount" or "operatorHost" &&
p.Value.ValueKind == JsonValueKind.String
                && p.Value.GetString() is string name && name.Length <= 100
&& name.All(c => char.IsLetterOrDigit(c) || "@._-\\.".Contains(c))
                && !OperationalRedaction.IsSensitive(name)) { safe[p.Name] =
name; }
        }
    }
    catch (JsonException) { return "{}"; }
    return JsonSerializer.Serialize(safe);
}

/// <summary>
/// Prompts for audit search criteria and displays sanitized results.
/// </summary>
public async Task<int> RunInteractiveAsync(Cancellation_token ct = default)
{
    if (Console.IsInputRedirected || Console.IsOutputRedirected) {
Console.Error.WriteLine("Use a private interactive terminal for scoped audit retrieval."); return
1; }

    Console.WriteLine("Read-only host operator audit retrieval. The API server will
not start. Output is sanitized JSON; copy only into approved private storage.");
    Console.Write("Database scope (PLATFORM or VENDOR): "); string? mode =
Console.ReadLine();
    if (mode is not ("PLATFORM" or "VENDOR")) { return 1; }
    Console.Write("Viewer account ID (current grants bound the export): "); if
(!Guid.TryParse(Console.ReadLine(), out Guid viewer)) { return 1; }
    Console.Write("Vendor ID (required for VENDOR, optional PLATFORM filter): ");
string? vendorText = Console.ReadLine(); Guid? vendor = Guid.TryParse(vendorText, out Guid v) ? v
: null;
    if (mode == "VENDOR" && vendor == null || !string.IsNullOrEmpty(vendorText) &&
vendor == null) { return 1; }
    Console.Write("From (ISO timestamp): "); if
(!DateTimeOffset.TryParse(Console.ReadLine(), out DateTimeOffset from)) { return 1; }
}

```

```

        Console.WriteLine("To (exclusive ISO timestamp, at most 31 days): "); if
(!DateTimeOffset.TryParse(Console.ReadLine(), out DateTimeOffset to)) { return 1; }
        Console.WriteLine("Actor ID filter (blank for all): "); string? actorText =
Console.ReadLine(); Guid? actor = Guid.TryParse(actorText, out Guid a) ? a : null; if
(!string.IsNullOrEmpty(actorText) && actor == null) { return 1; }
        Console.WriteLine("Resource ID filter (blank for all): "); string? resourceText =
Console.ReadLine(); Guid? resource = Guid.TryParse(resourceText, out Guid r) ? r : null; if
(!string.IsNullOrEmpty(resourceText) && resource == null) { return 1; }
        Console.WriteLine("Action filter (blank for all): "); string? action =
Console.ReadLine();
        Guid? customer = null, installation = null;
        if (mode == "VENDOR")
        {
            Console.WriteLine("Customer ID filter (blank for viewer's entire scope): ");
string? cText = Console.ReadLine(); customer = Guid.TryParse(cText, out Guid c) ? c : null;
            if (!string.IsNullOrEmpty(cText) && customer == null) { return 1; }
            Console.WriteLine("Installation ID filter (blank for viewer's entire scope):
"); string? iTText = Console.ReadLine(); installation = Guid.TryParse(iText, out Guid i) ? i :
null;

            if (!string.IsNullOrEmpty(iText) && installation == null) { return 1; }
        }
        Console.WriteLine("Outcome filter (blank for all): "); string? outcome =
Console.ReadLine();
        Console.WriteLine("Page (100 records maximum): "); if
(!int.TryParse(Console.ReadLine(), out int number)) { return 1; }
        AuditHistoryQuery filter = new AuditHistoryQuery(from, to, actor, resource,
string.IsNullOrEmpty(action) ? null : action, string.IsNullOrEmpty(outcome) ? null : outcome);
        PaginationQuery page = new PaginationQuery { Page = number, PageSize = 100 };
        PagedResponse<AuditHistoryRow> result = mode == "PLATFORM" ? await
PlatformAsync(viewer, filter, page, vendor, ct)
            : await VendorAsync(vendor!.Value, viewer, filter, page, customer,
installation, ct);
        Console.WriteLine(JsonSerializer.Serialize(result, new JsonSerializerOptions {
WriteIndented = true })); return 0;
    }
}

```

BoundedPatternKeyword.cs

```

using System.Text.Json;
using System.Text.RegularExpressions;
using BridgeControl.Api.Exceptions;
using Json.Schema.Keywords;

namespace BridgeControl.Services.Vendor;

/// <summary>Uses bounded, non-backtracking matching for the supported regular-expression
subset.</summary>
internal sealed class BoundedPatternKeyword : PatternKeyword
{
    /// <summary>
    /// Compiles a schema pattern with bounded regular-expression execution.
    /// </summary>
    internal static Regex Create(string pattern)
    {
        if (pattern.Length > 4096) { throw new ArtifactValidationException("Patterns are
limited to 4096 characters."); }
        try { return new Regex(pattern, RegexOptions.NonBacktracking |
RegexOptions.CultureInvariant, TimeSpan.FromMilliseconds(100)); }
        catch (Exception ex) when (ex is ArgumentException or NotSupportedException)

```

```

        { throw new ArtifactValidationException("Invalid or unsupported pattern. Use
regular expressions without backreferences or lookahead."); }
    }

    /// <summary>
    /// Validates and compiles the pattern keyword value.
    /// </summary>
    public override object? ValidateKeywordValue(JsonElement value) =>
Create(value.GetString()!);
}

```

ConfigurationArtifactReader.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOS.Vendor.DeploymentSettings;
using BridgeControl.Models.Vendor;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Reads archived configuration and installer artifacts with integrity checks.
/// </summary>
public sealed class ConfigurationArtifactReader(VendorArtifactArchive archive,
SchemaArtifactValidationService schemas)
{
    /// <summary>
    /// Reads product and upgrade identity from a verified archived installer.
    /// </summary>
    public async Task<InstallerIdentity> ReadInstallerIdentityAsync(string root, FileArtifact
artifact, CancellationToken ct)
    {
        await using Stream stream = await archive.OpenAsync(root, artifact, ct);
        return MsiArtifactValidator.ReadIdentity(((FileStream) stream).Name);
    }

    /// <summary>
    /// Reads a verified schema or configuration artifact as JSON.
    /// </summary>
    public async Task<JsonElement> ReadAsync(string root, FileArtifact artifact, bool schema,
CancellationToken ct)
    {
        if (artifact.SizeBytes > ConfigurationValues.MaximumBytes || artifact.SizeBytes <
1)
        { throw new ApiConflictException("artifact-unavailable", "The stored JSON artifact
is unavailable or invalid."); }
        try
        {
            await using Stream stream = await archive.OpenAsync(root, artifact, ct);
            using JsonDocument json = await JsonDocument.ParseAsync(stream, new() {
MaxDepth = 64 }, ct);
            if (schema) { schemas.Validate(json.RootElement, ct); }
            return json.RootElement.Clone();
        }
        catch (Exception error) when (error is IOException or InvalidOperationException or
JsonException or ArtifactValidationException)
        { throw new ApiConflictException("artifact-unavailable", "The stored JSON artifact
is unavailable or invalid."); }
    }
}

```

```

    /// <summary>
    /// Checks the archived installer against its recorded length and digest.
    /// </summary>
    public async Task VerifyInstallerAsync(string root, FileArtifact artifact,
CancellationToken ct)
    {
        try
        {
            await using Stream stream = await archive.OpenAsync(root, artifact, ct);
            MsiArtifactValidator.Validate(((FileStream)stream).Name);
            ct.ThrowIfCancellationRequested();
        }
        catch (Exception error) when (error is IOException or InvalidOperationException or
ArtifactValidationException)
        { throw new ApiConflictException("artifact-unavailable", "The stored installer is
unavailable or invalid."); }
    }
}

```

ConfigurationMigrationPlanner.cs

```

using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>Builds a side-effect-free candidate from exact source lineage and immutable rule
revisions.</summary>
public sealed class ConfigurationMigrationPlanner(ConfigurationValues values,
ConfigurationArtifactReader reader, ConfigurationMigrationRules rules)
{
    /// <summary>
    /// Builds a migration preview from version rules and current configuration layers.
    /// </summary>
    public async Task<ConfigurationMigrationPlan> PlanAsync(TenantContext tenant, Installation
installation, Guid targetId, Cancellation token ct)
    {
        VendorDbContext db = tenant.Database;
        EffectiveConfigurationRevision current = await
db.EffectiveConfigurationRevisions.Include(x => x.FileArtifact).Include(x => x.SourceRevisions)
        .SingleOrDefaultAsync(x => x.Id ==
installation.CurrentEffectiveConfigurationRevisionId && x.InstallationId == installation.Id, ct)
        ?? throw new ApiConflictException("migration-source-required", "Generate a
current effective configuration before previewing migration.");
        if (current.ProductVersionId == targetId) { throw new
ApiConflictException("migration-already-target", "The installation already targets this
version."); }
        MigrationVersionSnapshot source = await SnapshotAsync(tenant, installation,
current.ProductVersionId, ct);
        if (current.SchemaArtifactId != source.Schema.Id ||
!current.SourceRevisions.Select(x =>
x.ConfigurationRevisionId).Order().SequenceEqual(source.Layers.Where(x => x is not null).Select(x
=> x!.Id).Order()))

```

```

        { throw new ApiConflictException("migration-source-stale", "Source configuration
has changed. Wait for effective recalculation and preview again."); }
        JsonElement currentValues = await reader.ReadAsync(tenant.ArchivePath,
current.FileArtifact, false, ct);
        foreach (MigrationLayer? layer in source.Layers.Where(x => x is not null)) {
values.Validate(source.SchemaValues, layer!.Values, ct); }
        values.Validate(source.SchemaValues, currentValues, ct);
        if (!values.Merge(source.Layers.Where(x => x is not null).Select(x =>
x!.Values).ToArray()).SequenceEqual(values.Normalize(currentValues)))
        { throw new ApiConflictException("migration-source-inconsistent", "Current
effective configuration does not match its source layers."); }
        List<ConfigurationMigration> definitions = await
db.ConfigurationMigrations.Where(x => x.FromProductVersion.ProductId == installation.ProductId &&
x.ToProductVersion.ProductId == installation.ProductId)
        .OrderBy(x => x.Id).Take(257).ToListAsync(ct);
        if (definitions.Count > 256) { throw new
ApiConflictException("migration-graph-limit", "The migration graph exceeds the supported
bound."); }
        IReadOnlyList<ConfigurationMigration> path = SelectPath(definitions,
current.ProductVersionId, targetId);
        List<string> stamp = [$"installation:{installation.Id:N}",
$"current:{current.Id:N}", Stamp(source)];
        List<Guid> snapshots = []; List<MigrationValueChange> changes = []; List<string>
errors = [];
        JsonElement? customer = source.Layers[1]?.Values, local =
source.Layers[2]?.Values;
        MigrationVersionSnapshot target = source;
        JsonElement candidate = currentValues;
        int ruleCount = 0;
        foreach (ConfigurationMigration edge in path)
        {
            ct.ThrowIfCancellationRequested();
            MigrationDefinitionRevision definition = await
db.MigrationDefinitionRevisions.SingleOrDefaultAsync(x => x.MigrationId == edge.Id &&
x.RevisionNumber == edge.RevisionNumber, ct)
            ?? throw new
ApiConflictException("migration-definition-unverified", "Replace the legacy migration definition
before use.");
            MigrationRuleDto[] operations =
JsonSerializer.Deserialize<MigrationRuleDto[]>(definition.RulesJson) ?? throw new
ApiConflictException("migration-definition-invalid", "Stored rules are unavailable.");
            if ((ruleCount += operations.Length) > 256) { throw new
ApiConflictException("migration-path-limit", "A complete path supports at most 256 rules."); }
            rules.Validate(operations.Select(x => new MigrationRuleRequest(x.Sequence,
x.RuleType, x.SourcePath, x.TargetPath, x.Transform, x.IsDestructive)).ToArray());
            snapshots.Add(definition.Id); stamp.Add($"rules:{definition.Id:N}");
            target = await SnapshotAsync(tenant, installation, edge.ToProductVersionId,
ct); stamp.Add(Stamp(target));
            if (customer is JsonElement c) { MigrationLayerResult result =
rules.Apply(c, operations, "Customer"); customer = result.Values;
changes.AddRange(result.Changes); errors.AddRange(result.Unresolved); }
            if (local is JsonElement l) { MigrationLayerResult result = rules.Apply(l,
operations, "Installation"); local = result.Values; changes.AddRange(result.Changes);
errors.AddRange(result.Unresolved); }
            // Old defaults never become overrides: each hop uses its target version's
own default.
            try
            {
                using JsonDocument merged = JsonDocument.Parse(values.Merge(new
JsonElement?[] { target.Layers[0]?.Values, customer, local }.OfType<JsonElement>()).ToArray());
                candidate = merged.RootElement.Clone();

```

```

        if (target.Layers[0] is MigrationLayer defaults) {
values.Validate(target.SchemaValues, defaults.Values, ct); }
        if (customer is JsonElement c2) {
values.Validate(target.SchemaValues, c2, ct); }
        if (local is JsonElement l2) { values.Validate(target.SchemaValues,
12, ct); }

        values.Validate(target.SchemaValues, candidate, ct);
    }
    catch (ConfigurationValidationException invalid) { errors.Add($"Version
{edge.ToProductVersionId}: {invalid.Message}"); }
    }
    if (errors.Count == 0)
    {
        // A shared target override cannot be silently erased or replaced by
migration of another installation.
        CheckDestination(target.Layers[1], customer, "Customer", errors);
        CheckDestination(target.Layers[2], local, "Installation", errors);
    }
    string fingerprint =
Convert.ToHexStringLower(SHA256.HashData(Encoding.UTF8.GetBytes(string.Join('|', stamp))));
    return new(current.Id, source.Version.Id, targetId, target.Schema.Id, fingerprint,
candidate, customer, local,
        snapshots, changes, errors.Take(128).ToArray());
    }

    /// <summary>
    /// Detects existing target overrides that conflict with migrated values.
    /// </summary>
    private void CheckDestination(MigrationLayer? existing, JsonElement? migrated, string
layer, List<string> errors)
    {
        if (existing is not null && (migrated is null ||
!values.Normalize(existing.Values).SequenceEqual(values.Normalize(migrated.Value))))
        { errors.Add($"{{layer}}: existing target overrides differ from the migrated layer.
Resolve the target configuration before previewing again."); }
    }

    /// <summary>
    /// Selects a direct migration or a valid chain between the source and target versions.
    /// </summary>
    private static IReadOnlyList<ConfigurationMigration>
SelectPath(List<ConfigurationMigration> graph, Guid source, Guid target)
    {
        // Detect cycles reachable from this source, including malformed definitions
inserted outside the API.
        HashSet<Guid> visited = [], active = [];
        void Visit(Guid node, int depth)
        {
            if (active.Contains(node)) { throw new
ApiConflictException("migration-cycle", "The migration graph contains a cycle."); }
            if (visited.Contains(node)) { return; }
            if (depth > 256) { throw new ApiConflictException("migration-graph-limit",
"The migration graph is too deep."); }
            active.Add(node); foreach (ConfigurationMigration? edge in graph.Where(x =>
x.FromProductVersionId == node)) { Visit(edge.ToProductVersionId, depth + 1); }
            active.Remove(node); visited.Add(node);
        }
        Visit(source, 0);
        Queue<(Guid Version, List<ConfigurationMigration> Path)> queue = new();
        queue.Enqueue((source, [])); visited.Clear(); visited.Add(source);
        while (queue.TryDequeue(out (Guid Version, List<ConfigurationMigration> Path)
item))

```



```

        {
            foreach (ConfigurationMigration? edge in graph.Where(x =>
x.FromProductVersionId == item.Version).OrderBy(x => x.Id))
            {
                List<ConfigurationMigration> path = [.. item.Path, edge];
                if (edge.ToProductVersionId == target) { return path.Count <= 32 ?
path : throw new ApiConflictException("migration-path-limit", "Migration paths are limited to 32
steps."); }

                if (visited.Add(edge.ToProductVersionId)) {
queue.Enqueue((edge.ToProductVersionId, path)); }
            }

            throw new ApiConflictException("migration-path-missing", "No complete migration
path reaches the target version.");
        }

        /// <summary>
        /// Loads a version's schema and configuration layers for migration planning.
        /// </summary>
        private async Task<MigrationVersionSnapshot> SnapshotAsync(TenantContext tenant,
Installation installation, Guid versionId, CancellationToken ct)
        {
            VendorDbContext db = tenant.Database;
            ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x => x.Id
== versionId && x.ProductId == installation.ProductId, ct) ?? throw new ApiAccessException(404);
            if (version.Status is not ("Draft" or "Published")) { throw new
ApiConflictException("migration-version-unavailable", "A migration version is unavailable."); }
            Guid? schemaId = version.Status == "Published" ? version.PublishedSchemaArtifactId
: await db.ProductVersionArtifacts.Where(x => x.ProductVersionId ==
versionId && x.ArtifactType == "Schema").Select(x =>
(Guid?)x.FileArtifactId).SingleOrDefaultAsync(ct);
            FileArtifact schema = await db.FileArtifacts.SingleOrDefaultAsync(x => x.Id ==
schemaId && x.ArtifactType == "Schema", ct)
            ?? throw new ApiConflictException("schema-required", "Every migration
version requires a valid schema.");
            List<ConfigurationProfile> profiles = await db.ConfigurationProfiles.Where(x =>
(x.ProductId == installation.ProductId && x.InstallationId == null && (x.CustomerId == null ||
x.CustomerId == installation.CustomerId))
            || (x.InstallationId == installation.Id && x.ProductId == null &&
x.CustomerId == null)).ToListAsync(ct);
            MigrationLayer?[] layers = new MigrationLayer?[3];
            for (int i = 0; i < 3; i++)
            {
                ConfigurationProfile? profile = profiles.SingleOrDefault(x => i == 0 ?
x.ProductId != null && x.CustomerId == null : i == 1 ? x.CustomerId == installation.CustomerId :
x.InstallationId == installation.Id);
                ConfigurationRevision? revision = null;
                if (i == 0 && version.Status == "Published")
                {
                    if (version.PublishedDefaultRevisionId is Guid pinned)
                    {
                        revision = await db.ConfigurationRevisions.Include(x =>
x.FileArtifact).SingleOrDefaultAsync(x => x.Id == pinned, ct);
                        if (revision is null || profile is null ||
revision.ConfigurationProfileId != profile.Id || revision.ProductVersionId != versionId)
                        { throw new
ApiConflictException("migration-source-ownership", "The published default has invalid
ownership."); }
                    }
                    else if (profile is not null)

```

```

        { revision = await db.ConfigurationRevisions.Include(x =>
x.FileArtifact).Where(x => x.ConfigurationProfileId == profile.Id && x.ProductVersionId ==
versionId).OrderByDescending(x => x.RevisionNumber).FirstOrDefaultAsync(ct); }
        if (revision is not null)
        {
            if (revision.FileArtifact.ArtifactType != "Configuration") { throw
new ApiConflictException("migration-source-invalid", "A source artifact is invalid."); }
            layers[i] = new(revision.Id, await
reader.ReadAsync(tenant.ArchivePath, revision.FileArtifact, false, ct));
        }
    }
    return new(version, schema, await reader.ReadAsync(tenant.ArchivePath, schema,
true, ct), layers);
}

/// <summary>
/// Creates a comparison stamp from the version, schema, and layer identities.
/// </summary>
private static string Stamp(MigrationVersionSnapshot snapshot) =>
$"{snapshot.Version.Id:N}:{snapshot.Version.Status}:{snapshot.Schema.Id:N}:" + string.Join(',',
snapshot.Layers.Select(x => x?.Id.ToString("N") ?? "absent"));
}

/// <summary>
/// Represents configuration migration plan data.
/// </summary>
public sealed record ConfigurationMigrationPlan(Guid SourceRevisionId, Guid FromProductVersionId,
Guid TargetProductVersionId, Guid SchemaArtifactId,
    string Fingerprint, JsonElement Values, JsonElement? CustomerValues, JsonElement?
InstallationValues,
    IReadOnlyList<Guid> DefinitionRevisionIds, IReadOnlyList<MigrationValueChange> Changes,
IReadOnlyList<string> Errors);

/// <summary>
/// Represents migration layer data.
/// </summary>
internal sealed record MigrationLayer(Guid Id, JsonElement Values);

/// <summary>
/// Represents migration version snapshot data.
/// </summary>
internal sealed record MigrationVersionSnapshot(ProductVersion Version, FileArtifact Schema,
JsonElement SchemaValues, MigrationLayer?[] Layers);

```

ConfigurationMigrationRules.cs

```

using System.Globalization;
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using System.Text.Json.Nodes;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOS.Vendor;

namespace BridgeControl.Services.Vendor;

/// <summary>Bounded declarative transformations. Paths address object properties; arrays are
atomic values.</summary>
public sealed class ConfigurationMigrationRules(ConfigurationValues values)
{

```

```

    /// <summary>
    /// Checks rule operations, paths, and transformations before a migration is saved.
    /// </summary>
    public void Validate(IReadOnlyList<MigrationRuleRequest> rules)
    {
        if (rules is null || rules.Count > 64 || rules.Any(x => x is null) ||
rules.Select(x => x.Sequence).Distinct().Count() != rules.Count)
        { throw new ConfigurationValidationException("Supply at most 64 rules with unique
sequences."); }
        foreach (MigrationRuleRequest rule in rules)
        {
            if (rule.Sequence < 1 || rule.RuleType is not ("Preserve" or "Move" or
"Rename" or "Convert" or "Introduce" or "Delete"))
            { throw new ConfigurationValidationException("Use positive sequences and a
supported rule type."); }
            bool source = rule.RuleType != "Introduce", target = rule.RuleType is
"Move" or "Rename" or "Introduce";
            if (source) { Path(rule.SourcePath); } else if
(!string.IsNullOrEmpty(rule.SourcePath)) { Invalid(); }
            if (target) { Path(rule.TargetPath); } else if
(!string.IsNullOrEmpty(rule.TargetPath)) { Invalid(); }
            if (rule.RuleType is "Move" or "Rename")
            {
                if (rule.SourcePath == rule.TargetPath ||
rule.SourcePath!.StartsWith(rule.TargetPath + "/", StringComparison.Ordinal)
|| rule.TargetPath!.StartsWith(rule.SourcePath + "/",
StringComparison.Ordinal)) { Invalid(); }
            }
            if (rule.RuleType == "Convert")
            { if (rule.Transform is not ("string" or "integer" or "number" or
"boolean")) { Invalid(); } }
            else if (rule.RuleType == "Introduce")
            {
                try
                {
                    using JsonDocument doc = JsonDocument.Parse(rule.Transform
?? ""); if (doc.RootElement.ValueKind == JsonValueKind.Null) { Invalid(); }
                    values.Normalize(JsonSerializer.SerializeToElement(new {
value = doc.RootElement }));
                }
                catch (JsonException) { throw new
ConfigurationValidationException("Introduce requires a bounded non-null JSON literal."); }
            }
            else if (!string.IsNullOrEmpty(rule.Transform)) { Invalid(); }
            if (rule.RuleType is "Delete" or "Convert" && !rule.IsDestructive) { throw
new ConfigurationValidationException("Delete and Convert require IsDestructive=true."); }
        }
    }

    /// <summary>
    /// Rejects an invalid migration rule with a configuration validation error.
    /// </summary>
    private static void Invalid() => throw new ConfigurationValidationException("Rule paths or
transformation do not match the operation.");

    /// <summary>
    /// Parses and validates an object-property JSON pointer.
    /// </summary>
    private static string[] Path(string? pointer)
    {
        if (string.IsNullOrEmpty(pointer) || pointer.Length > 1024 ||
!pointer.StartsWith('/') || pointer.Split('/').Length > 33)

```

```

        { throw new ConfigurationValidationException("Use a non-root JSON Pointer of at
most 1024 characters and 32 segments."); }
        string[] parts = pointer[1..].Split('/');
        foreach (string part in parts)
        {
            for (int i = 0; i < part.Length; i++)
            { if (part[i] == '~' && (++i == part.Length || part[i] is not ('0' or
'1')) { throw new ConfigurationValidationException("Invalid JSON Pointer escape."); } }
            return parts.Select(x => x.Replace("~1", "/",
StringComparison.Ordinal).Replace("~0", "~", StringComparison.Ordinal)).ToArray();
        }

    /// <summary>
    /// Applies ordered migration rules to a layer and records changes and unresolved values.
    /// </summary>
    public MigrationLayerResult Apply(JsonElement input, IReadOnlyList<MigrationRuleDto>
rules, string layer)
    {
        JsonObject root = JsonNode.Parse(values.Normalize(input)).AsObject();
        List<MigrationValueChange> changes = []; List<string> unresolved = [];
        foreach (MigrationRuleDto? rule in rules.OrderBy(x => x.Sequence))
        {
            string pointer = rule.RuleType == "Introduce" ? rule.TargetPath :
rule.SourcePath;
            string[] path = Path(pointer);
            if (CrossesAtomicValue(root, path)) { unresolved.Add($"{layer}: {pointer}
traverses an atomic value."); continue; }
            JsonObject? parent = Parent(root, path, false); bool exists =
parent?.ContainsKey(path[^1]) == true;
            JsonNode? before = exists ? parent![path[^1]]?.DeepClone() : null;
            if (rule.RuleType != "Introduce" && !exists) { continue; }
            try
            {
                switch (rule.RuleType)
                {
                    case "Preserve": break;
                    case "Delete": parent!.Remove(path[^1]); break;
                    case "Convert": parent![path[^1]] = ConvertValue(before,
rule.Transform); break;
                    case "Move":
                    case "Rename":
                        string[] destination = Path(rule.TargetPath);
                        JsonObject? target = Parent(root, destination,
true);
                        if (target is null ||
target.ContainsKey(destination[^1])) { throw new InvalidOperationException(); }
                        target[destination[^1]] = before?.DeepClone();
                        parent!.Remove(path[^1]); break;
                    case "Introduce":
                        JsonNode? introduced =
JsonNode.Parse(rule.Transform);
                        if (exists && !JsonNode.DeepEquals(before,
introduced)) { throw new InvalidOperationException(); }
                        parent = Parent(root, path, true) ?? throw new
InvalidOperationException(); parent[path[^1]] = introduced; break;
                    default: throw new
ConfigurationValidationException("Unsupported stored migration rule.");
                }
                // Enforce configuration bounds after each operation, including
newly introduced subtrees.
                values.Normalize(JsonSerializer.SerializeToElement(root));
            }
        }
    }

```

```

        changes.Add(new(rule.Id, layer, rule.RuleType, rule.SourcePath,
rule.TargetPath, rule.IsDestructive,
        before is null ? null :
Convert.ToHexStringLower(SHA256.HashData(Encoding.UTF8.GetBytes(before.ToJsonString()))));
    }
    catch (Exception error) when (error is InvalidOperationException or
FormatException or OverflowException)
    { unresolved.Add($"{{layer}}: {{pointer}} cannot be migrated by rule
{{rule.Id}}."); }
    }
    return new(JsonSerializer.SerializeToElement(root), changes, unresolved);
}

/// <summary>
/// Locates a property's parent object, optionally creating missing objects.
/// </summary>
private static JsonObject? Parent(JsonObject root, string[] path, bool create)
{
    JsonObject parent = root;
    foreach (string segment in path[..^1])
    {
        if (!parent.ContainsKey(segment) && create) { parent[segment] = new
JsonObject(); }

        if (parent[segment] is not JsonObject child) { return null; }
        parent = child;
    }
    return parent;
}

/// <summary>
/// Checks whether a path traverses a scalar or array instead of an object.
/// </summary>
private static bool CrossesAtomicValue(JsonObject root, string[] path)
{
    JsonObject parent = root;
    foreach (string segment in path[..^1])
    {
        if (!parent.ContainsKey(segment)) { return false; }
        if (parent[segment] is not JsonObject child) { return true; }
        parent = child;
    }
    return false;
}

/// <summary>
/// Converts a JSON value using the requested supported transformation.
/// </summary>
private static JsonNode ConvertValue(JsonNode? node, string transform)
{
    if (node is not JsonValue value) { throw new InvalidOperationException(); }
    string text = value.TryGetValue<string>(out string? str) ? str :
value.ToJsonString();
    return transform switch
    {
        "string" => JsonValue.Create(text)!,
        "integer" when long.TryParse(text, NumberStyles.AllowLeadingSign,
CultureInfo.InvariantCulture, out long number) && Math.Abs((decimal)number) <= 9007199254740991
=> JsonValue.Create(number)!,
        "number" when double.TryParse(text, NumberStyles.Float,
CultureInfo.InvariantCulture, out double number) && double.IsFinite(number) =>
JsonValue.Create(number)!,
    }
}

```

```

        "boolean" when text is "true" or "false" => JsonValue.Create(text ==
"true")!,
        _ => throw new FormatException()
    };
}

}

/// <summary>
/// Represents migration value change data.
/// </summary>
public sealed record MigrationValueChange(Guid RuleId, string Layer, string RuleType, string
SourcePath, string TargetPath, bool IsDestructive, string? PreviousValueHash);

/// <summary>
/// Represents migration layer result data.
/// </summary>
public sealed record MigrationLayerResult(JsonElement Values, IReadOnlyList<MigrationValueChange>
Changes, IReadOnlyList<string> Unresolved);

```

ConfigurationMigrationService.cs

```

using System.Security.Cryptography;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOS.Vendor;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>Internal review and commit operations; approval identity is selected by the API
adapter.</summary>
public sealed class ConfigurationMigrationService(TenantOperationFactory operations,
ConfigurationMigrationPlanner planner, ConfigurationValues values,
    ConfigurationArtifactReader reader, VendorArtifactArchive archive,
EffectiveConfigurationGenerator generator, VendorSecurityAudit audit, TimeProvider clock)
{
    /// <summary>
    /// Saves a migration preview with the source evidence required for approval.
    /// </summary>
    public async Task<ConfigurationMigrationReview> ReviewAsync(TenantActor actor, Guid
installationId, Guid targetId, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.manage", true, ct);
        await using IDbContextTransaction transaction = await
op.Database.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Installation installation = await InstallationAsync(op, installationId, targetId,
ct);
        ConfigurationMigrationPlan plan = await planner.PlanAsync(op.Tenant, installation,
targetId, ct);
        // The mutation lock serializes reviews and apply; monotonic timestamps make
"latest" unambiguous even with a frozen/coarse clock.
        DateTimeOffset createdAt = clock.GetUtcNow();
        DateTimeOffset? latest = await op.Database.ConfigurationMigrationReviews.Where(x
=> x.UserId == actor.UserId && x.InstallationId == installationId && x.TargetProductVersionId ==
targetId)
            .MaxAsync(x => (DateTimeOffset?)x.CreatedAt, ct);
    }
}

```

```

        if (latest >= createdAt) { createdAt = latest.Value.AddTicks(1); }
        ConfigurationMigrationReview review = new ConfigurationMigrationReview
        {
            UserId = actor.UserId,
            InstallationId = installation.Id,
            TargetProductVersionId = targetId,
            Fingerprint = plan.Fingerprint,
            PlanJson = JsonSerializer.Serialize(plan),
            CreatedAt = createdAt
        };
        op.Database.ConfigurationMigrationReviews.Add(review);
        audit.Record(op.Database, actor.UserId, "ConfigurationMigrationPreviewed",
installation.Id, entityType: "Installation", resourceId: review.Id);
        await op.Database.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return
review;
    }

    /// <summary>
    /// Applies the selected migration review after rechecking its approval and evidence.
    /// </summary>
    public Task<EffectiveConfigurationResponse> ApplyReviewedAsync(TenantActor actor, Guid
installationId, Guid targetId, Guid reviewId, IReadOnlyList<Guid> approvedRuleIds,
CancellationToken ct)
    => ApplyCoreAsync(actor, installationId, targetId, reviewId, approvedRuleIds, ct);

    /// <summary>
    /// Applies the current user's latest valid review for the requested target.
    /// </summary>
    public Task<EffectiveConfigurationResponse> ApplyLatestAsync(TenantActor actor, Guid
installationId, Guid targetId, IReadOnlyList<Guid> approvedRuleIds, CancellationToken ct)
    => ApplyCoreAsync(actor, installationId, targetId, null, approvedRuleIds, ct);

    /// <summary>
    /// Validates migration evidence and atomically writes the migrated configuration layers.
    /// </summary>
    private async Task<EffectiveConfigurationResponse> ApplyCoreAsync(TenantActor actor, Guid
installationId, Guid targetId, Guid? reviewId, IReadOnlyList<Guid> approvedRuleIds,
CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"configuration.manage", true, ct);
        VendorDbContext db = op.Database;
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct);
        Installation installation = await InstallationAsync(op, installationId, targetId,
ct);

        IQueryable<ConfigurationMigrationReview> reviews =
db.ConfigurationMigrationReviews.Where(x => x.UserId == actor.UserId && x.InstallationId ==
installationId && x.TargetProductVersionId == targetId);
        if (reviewId is Guid id) { reviews = reviews.Where(x => x.Id == id); }
        ConfigurationMigrationReview review = await reviews.OrderByDescending(x =>
x.CreatedAt).ThenByDescending(x => x.Id).FirstOrDefaultAsync(ct)
        ?? throw new ApiConflictException("migration-preview-required", "Preview
this migration with the current user before applying it.");
        if (approvedRuleIds is null || approvedRuleIds.Count > 256 ||
approvedRuleIds.Distinct().Count() != approvedRuleIds.Count)
        { throw new ConfigurationValidationException("Supply a unique list of at most 256
approved destructive rule IDs."); }
        string approvals = JsonSerializer.Serialize(approvedRuleIds.Order().ToArray());
        if (review.ResultRevisionId is Guid existing)
        {

```

```

        if (review.ApprovedRuleIdsJson != approvals) { throw new
ApiConflictException("migration-approval-mismatch", "Retry with the original approvals."); }
        EffectiveConfigurationResponse response = await ResponseAsync(op, existing,
ct); await transaction.CommitAsync(ct); return response;
    }
    if (review.CreatedAt.AddMinutes(30) <= clock.UtcNow()) { throw new
ApiConflictException("migration-preview-expired", "The preview has expired. Preview again."); }
    ConfigurationMigrationPlan reviewed =
JsonSerializer.Deserialize<ConfigurationMigrationPlan>(review.PlanJson!);
    if (reviewed.Errors.Count > 0) { throw new
ApiConflictException("migration-unresolved", "Resolve the preview's errors and preview again
before applying."); }
    ConfigurationMigrationPlan plan = await planner.PlanAsync(op.Tenant, installation,
targetId, ct);
    if (plan.Fingerprint != review.Fingerprint) { throw new
ApiConflictException("migration-preview-stale", "Source, target or rules changed after preview.
Preview again."); }
    if (plan.Errors.Count > 0) { throw new
ApiConflictException("migration-unresolved", "The migration still contains unresolved values.");
    }
    Guid[] required = plan.Changes.Where(x => x.IsDestructive).Select(x =>
x.RuleId).Distinct().Order().ToArray();
    if (!required.SequenceEqual(approvedRuleIds.Order())) { throw new
ApiConflictException("migration-approval-required", "Approve exactly the destructive rule IDs
reported by this preview."); }
    await WriteLayerAsync(op, actor, installation, plan, true, plan.CustomerValues,
ct);
    await WriteLayerAsync(op, actor, installation, plan, false,
plan.InstallationValues, ct);
    await db.SaveChangesAsync(ct);
    Guid resultId = await generator.GenerateLockedAsync(op.Tenant, installation.Id,
targetId, true, ct);
    EffectiveConfigurationResponse result = await ResponseAsync(op, resultId, ct);
    if (!values.Normalize(result.Values).SequenceEqual(values.Normalize(plan.Values)))
    { throw new ApiConflictException("migration-result-changed", "The generated result
differs from the reviewed candidate."); }
    review.ResultRevisionId = resultId; review.AppliedAt = clock.UtcNow();
review.ApprovedRuleIdsJson = approvals;
    dbVendorAuditLogs.Add(new VendorAuditLog
    {
        ActorId = actor.UserId,
        ActorType = "User",
        Action = "ConfigurationMigrationApplied",
        EntityId = installationId,
        EntityType = "Installation",
        CreatedAt = clock.UtcNow(),
        Context = JsonSerializer.Serialize(new { reviewId = review.Id,
sourceRevisionId = plan.SourceRevisionId, resultRevisionId = resultId })
    });
    for (int index = 0; index < plan.Changes.Count; index++)
    {
        MigrationValueChange change = plan.Changes[index]; if
(!change.IsDestructive) { continue; }
        // Exact paths, rule snapshots and pre-change hashes are retained in the
review; keep audit rows small and free of raw configuration values.
        dbVendorAuditLogs.Add(new VendorAuditLog
        {
            ActorId = actor.UserId,
            ActorType = "User",
            Action = "ConfigurationMigrationDestructiveChangeApproved",
            EntityId = installationId,
            EntityType = "Installation",

```



```

        CreatedAt = clock.UtcNow(),
        Context = JsonSerializer.Serialize(new { reviewId = review.Id,
changeIndex = index, ruleId = change.RuleId, layer = change.Layer, previousValueHash =
change.PreviousValueHash })
    });
    }
    await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct); return result;
}

/// <summary>
/// Loads an installation within the current actor's migration scope.
/// </summary>
private static async Task<Installation> InstallationAsync(TenantOperation op, Guid
installation, Guid target, CancellationToken ct)
{
    Installation item = await op.Access.OperationalInstallations(op.Database,
"configuration.manage").SingleOrDefaultAsync(x => x.Id == installation, ct) ?? throw new
ApiAccessException(404);
    if (!await op.Database.ProductVersions.AnyAsync(x => x.Id == target && x.ProductId
== item.ProductId, ct)) { throw new ApiAccessException(404); }
    return item;
}

/// <summary>
/// Writes a migrated configuration layer and its immutable revision history.
/// </summary>
private async Task WriteLayerAsync(TenantOperation op, TenantActor actor, Installation
installation, ConfigurationMigrationPlan plan, bool customer, JsonElement? content,
CancellationToken ct)
{
    if (content is not JsonElement json) { return; }
    VendorDbContext db = op.Database;
    ConfigurationProfile? profile = await
db.ConfigurationProfiles.SingleOrDefaultAsync(x => customer
? x.CustomerId == installation.CustomerId && x.ProductId ==
installation.ProductId && x.InstallationId == null
: x.InstallationId == installation.Id && x.CustomerId == null &&
x.ProductId == null, ct);
    if (profile is null)
    {
        profile = new()
        {
            Id = Guid.NewGuid(),
            ProductId = customer ? installation.ProductId : null,
            CustomerId = customer ? installation.CustomerId : null,
            InstallationId = customer ? null : installation.Id,
            CreatedAt = clock.UtcNow()
        }; db.ConfigurationProfiles.Add(profile);
    }
    byte[] canonical = values.Normalize(json); string hash =
Convert.ToHexStringLower(SHA256.HashData(canonical));
    ConfigurationRevision? existing = await db.ConfigurationRevisions.Include(x =>
x.FileArtifact).Where(x => x.ConfigurationProfileId == profile.Id && x.ProductVersionId ==
plan.TargetProductVersionId).OrderByDescending(x => x.RevisionNumber).FirstOrDefaultAsync(ct);
    if (existing is not null && existing.SchemaArtifactId == plan.SchemaArtifactId &&
existing.ValidationStatus == "Valid" && existing.FileArtifact.Sha256 == hash &&
existing.FileArtifact.SizeBytes == canonical.Length)
    { await reader.ReadAsync(op.Tenant.ArchivePath, existing.FileArtifact, false, ct);
return; }
    await using StagedArtifact staged = await
archive.StageConfigurationAsync(op.Tenant.ArchivePath, canonical, ct);
    ConfigurationRevision revision = new ConfigurationRevision

```

```

        {
            Id = Guid.NewGuid(),
            ConfigurationProfileId = profile.Id,
            ProductVersionId = plan.TargetProductVersionId,
            SchemaArtifactId = plan.SchemaArtifactId,
            ValidationStatus = "Valid",
            RevisionNumber = checked(++profile.LastRevisionNumber),
            CreatedByUserId = actor.UserId,
            CreatedAt = clock.UtcNow(),
            FileArtifact = new FileArtifact
            {
                Id = Guid.NewGuid(),
                ArtifactType = "Configuration",
                FileName = staged.FileName,
                ContentType = staged.ContentType,
                StorageIdentifier = staged.StorageIdentifier,
                Sha256 = staged.Sha256,
                SizeBytes = staged.SizeBytes,
                CreatedAt = clock.UtcNow()
            }
        };
        db.ConfigurationRevisions.Add(revision); db.ConfigurationSourceChanges.Add(new() {
Id = Guid.NewGuid(), ConfigurationRevisionId = revision.Id, CreatedAt = clock.UtcNow() });
        audit.Record(db, actor.UserId, "ConfigurationMigrated", profile.Id, entityType:
"ConfigurationProfile", resourceId: revision.Id); staged.Promote();
    }

    /// <summary>
    /// Builds the migration result from the committed configuration state.
    /// </summary>
    private async Task<EffectiveConfigurationResponse> ResponseAsync(TenantOperation op, Guid
id, CancellationToken ct)
    {
        EffectiveConfigurationRevision revision = await
op.Database.EffectiveConfigurationRevisions.Include(x => x.FileArtifact).Include(x =>
x.SourceRevisions).SingleAsync(x => x.Id == id, ct);
        return new(revision.Id, revision.InstallationId, revision.ProductVersionId,
revision.RevisionNumber,
            await reader.ReadAsync(op.Tenant.ArchivePath, revision.FileArtifact, false,
ct), revision.SourceRevisions.Select(x => x.ConfigurationRevisionId).Order().ToArray(),
revision.CreatedAt);
    }
}

```

ConfigurationValues.cs

```

using System.Text;
using System.Text.Json;
using System.Text.Json.Nodes;
using System.Text.RegularExpressions;
using BridgeControl.Api.Exceptions;
using Json.Schema;
using Org.Webpki.JsonCanonicalizer;

namespace BridgeControl.Services.Vendor;

/// <summary>Configuration cleanup, RFC 8785 bytes and bounded evaluation of the accepted schema
profile.</summary>
public sealed class ConfigurationValues(SchemaArtifactValidationService schemas)
{

```

```

public const int MaximumBytes = 1024 * 1024;

/// <summary>
/// Checks configuration bounds and returns cleaned, canonical JSON bytes.
/// </summary>
public byte[] Normalize(JsonElement input)
{
    if (input.ValueKind != JsonValueKind.Object) { throw new
ConfigurationValidationException("Values must be an object."); }
    if (Encoding.UTF8.GetByteCount(input.GetRawText()) > MaximumBytes) { throw new
ConfigurationValidationException("Configuration exceeds 1 MiB."); }
    int count = 0;
    try { Check(input, 0, ref count); }
    catch (InvalidOperationException) { throw new
ConfigurationValidationException("Values contain invalid Unicode or JSON data."); }
    using MemoryStream output = new();
    using (Utf8JsonWriter writer = new(output)) { WriteClean(writer, input); }
    try { return new JsonCanonicalizer(output.ToArray()).GetEncodedUTF8(); }
    catch (IOException) { throw new ConfigurationValidationException("Values cannot be
represented as canonical JSON."); }
}

/// <summary>
/// Validates configuration values against the supported schema with bounded evaluation
work.
/// </summary>
public void Validate(JsonElement schema, JsonElement values, CancellationToken ct)
{
    schemas.Validate(schema, ct);
    if (values.ValueKind != JsonValueKind.Object) { throw new
ConfigurationValidationException("Values must be an object."); }
    if (Encoding.UTF8.GetByteCount(values.GetRawText()) > MaximumBytes) { throw new
ConfigurationValidationException("Configuration exceeds 1 MiB."); }
    int count = 0;
    try { Check(values, 0, ref count); }
    catch (InvalidOperationException) { throw new
ConfigurationValidationException("Values contain invalid Unicode or JSON data."); }
    // Bound expanded reference/applicator work before entering the synchronous schema
library.

    long work = ExpandedCost(schema, schema, values, [], 0, ct);
    ct.ThrowIfCancellationRequested();
    DialectRegistry dialects = new();
    dialects.Register(Dialect.Draft202012.With([new BoundedPatternKeyword()], id: new
Uri(Draft202012.SchemaArtifactValidator.DialectUri)));
    BuildOptions options = new() { SchemaRegistry = new SchemaRegistry { Fetch = (_,
_) => null }, DialectRegistry = dialects };
    JsonSchema compiled = JsonSchema.FromText(schema.GetRawText(), options);
    EvaluationResults result = Evaluate(compiled, values, OutputFormat.Flag);
    ct.ThrowIfCancellationRequested();
    if (!result.IsValid)
    {
        if (work > 100_000) { throw new ConfigurationValidationException("Values do
not satisfy the selected schema at /; detailed diagnostics exceed the diagnostic work limit."); }
        result = Evaluate(compiled, values, OutputFormat.List);
        // Return bounded JSON-pointer/keyword locations, never library messages
containing submitted values.
        IEnumerable<string> errors = new[] { result }.Concat(result.Details ??
[]).Where(x => !x.IsValid && x.Errors?.Count > 0)
            .Take(8).Select(x => $"{Bound(x.InstanceLocation.ToString())}
({string.Join(",", x.Errors!.Keys.Take(4))})");
        throw new ConfigurationValidationException("Values do not satisfy the
selected product version's schema: " + string.Join(" ", errors));
    }
}

```

```

    }
}

/// <summary>
/// Limits a diagnostic JSON pointer and represents the root as a slash.
/// </summary>
private static string Bound(string location) => location.Length == 0 ? "/" :
location[..Math.Min(160, location.Length)];

/// <summary>
/// Overlays configuration layers in order and returns canonical merged JSON.
/// </summary>
public byte[] Merge(params JsonElement[] layers)
{
    JsonObject merged = [];
    foreach (JsonElement layer in layers)
    {
        JsonObject cleaned = JsonNode.Parse(Normalize(layer))!.AsObject();
        MergeObjects(merged, cleaned);
    }
    return Normalize(JsonSerializer.SerializeToElement(merged));
}

/// <summary>
/// Recursively merges objects while replacing arrays and scalar values.
/// </summary>
private static void MergeObjects(JsonObject target, JsonObject source)
{
    foreach (KeyValuePair<string, JsonNode?> pair in source)
    {
        if (pair.Value is JsonObject incoming && target[pair.Key] is JsonObject
existing) { MergeObjects(existing, incoming); }
        else { target[pair.Key] = pair.Value?.DeepClone(); }
    }
}

/// <summary>
/// Checks JSON size, depth, property uniqueness, and supported scalar values.
/// </summary>
private static void Check(JsonElement value, int depth, ref int count)
{
    if (++count > 4096 || depth > 32) { throw new
ConfigurationValidationException("Values exceed the 4096-node or depth-32 limit."); }
    switch (value.ValueKind)
    {
        case JsonValueKind.Object:
            HashSet<string> names = [];
            foreach (JsonProperty property in value.EnumerateObject())
            {
                CheckString(property.Name);
                if (!names.Add(property.Name)) { throw new
ConfigurationValidationException("Duplicate JSON keys are not allowed."); }
                Check(property.Value, depth + 1, ref count);
            }
            break;
        case JsonValueKind.Array:
            if (value.GetArrayLength() > 256) { throw new
ConfigurationValidationException("Arrays are limited to 256 entries."); }
            foreach (JsonElement item in value.EnumerateArray()) { Check(item,
depth + 1, ref count); }
            break;
        case JsonValueKind.String: CheckString(value.GetString()); break;
    }
}

```

```

        case JsonValueKind.Number:
            if (!value.TryGetDouble(out double number) ||
!double.IsFinite(number)
                || (Math.Truncate(number) == number && Math.Abs(number) >
9007199254740991D))
            { throw new ConfigurationValidationException("Numbers must be
finite; integers outside the exact JSON range must be strings."); }
            break;
        }
    }

    /// <summary>
    /// Rejects unpaired Unicode surrogates in configuration strings.
    /// </summary>
    private static void CheckString(string value)
    {
        for (int i = 0; i < value.Length; i++)
        {
            if (char.IsHighSurrogate(value[i]) && i + 1 < value.Length &&
char.IsLowSurrogate(value[i + 1])) { i++; continue; }
            if (char.IsSurrogate(value[i])) { throw new
ConfigurationValidationException("Invalid Unicode is not allowed."); }
        }
    }

    /// <summary>
    /// Writes configuration JSON while omitting object properties with null values.
    /// </summary>
    private static void WriteClean(Utf8JsonWriter writer, JsonElement value)
    {
        if (value.ValueKind == JsonValueKind.Object)
        {
            writer.WriteStartObject();
            foreach (JsonProperty property in value.EnumerateObject())
            { if (property.Value.ValueKind != JsonValueKind.Null) {
writer.WritePropertyName(property.Name); WriteClean(writer, property.Value); } }
            writer.WriteEndObject();
        }
        else if (value.ValueKind == JsonValueKind.Array)
        { writer.WriteStartArray(); foreach (JsonElement item in value.EnumerateArray()) {
WriteClean(writer, item); } writer.WriteEndArray(); }
        else { value.WriteTo(writer); }
    }

    /// <summary>
    /// Evaluates a compiled schema and translates bounded-pattern failures into validation
errors.
    /// </summary>
    private static EvaluationResults Evaluate(JsonSchema compiled, JsonElement values,
OutputFormat output)
    {
        try { return compiled.Evaluate(values, new EvaluationOptions { OutputFormat =
output, RequireFormatValidation = true }); }
        catch (RegexMatchTimeoutException) { throw new
ConfigurationValidationException("Pattern matching exceeded the validation time limit."); }
    }

    /// <summary>
    /// Bounds the work required by schema references and applicators for the supplied values.
    /// </summary>
    private static long ExpandedCost(JsonElement node, JsonElement root, JsonElement value,

```

```

        Dictionary<(JsonElement Schema, JsonElement Value), long> memo, int depth,
CancellationToken ct)
    {
        ct.ThrowIfCancellationRequested();
        if (depth > 2304) { throw new ConfigurationValidationException("Schema expansion
is too deep."); }
        if (memo.TryGetValue((node, value), out long cached)) { return cached; }
        long cost = 1;
        void Add(JsonElement child, JsonElement instance)
        {
            cost += ExpandedCost(child, root, instance, memo, depth + 1, ct);
            if (cost > 2_000_000) { throw new ConfigurationValidationException("Schema
expansion exceeds the validation work limit."); }
        }
        if (node.ValueKind != JsonValueKind.Object) { return cost; }
        foreach (JsonProperty keyword in node.EnumerateObject())
        {
            if (keyword.Name == "$ref")
            {
                JsonElement target = root;
                foreach (string segment in
keyword.Value.GetString()[2..].Split('/'))
                {
                    string key = segment.Replace("~1", "/").Replace("~0", "~");
                    target = target.ValueKind == JsonValueKind.Array ?
target[int.Parse(key, System.Globalization.CultureInfo.InvariantCulture)] :
target.GetProperty(key);
                }
                Add(target, value);
            }
            else if (keyword.Name == "properties" && value.ValueKind ==
JsonValueKind.Object)
            {
                foreach (JsonProperty child in keyword.Value.EnumerateObject())
                { if (value.TryGetProperty(child.Name, out JsonElement instance)) {
Add(child.Value, instance); } }
            }
            else if (keyword.Name == "additionalProperties" && value.ValueKind ==
JsonValueKind.Object)
            {
                node.TryGetProperty("properties", out JsonElement properties);
                foreach (JsonProperty child in value.EnumerateObject())
                { if (properties.ValueKind != JsonValueKind.Object ||
!properties.TryGetProperty(child.Name, out _)) { Add(keyword.Value, child.Value); } }
            }
            else if (keyword.Name == "items" && value.ValueKind == JsonValueKind.Array)
            {
                int prefix = node.TryGetProperty("prefixItems", out JsonElement
tuple) ? tuple.GetArrayLength() : 0;
                foreach (JsonElement item in value.EnumerateArray().Skip(prefix)) {
Add(keyword.Value, item); }
            }
            else if (keyword.Name == "prefixItems" && value.ValueKind ==
JsonValueKind.Array)
            {
                for (int i = 0; i < Math.Min(value.GetArrayLength(),
keyword.Value.GetArrayLength()); i++) { Add(keyword.Value[i], value[i]); }
            }
            else if (keyword.Name is "not" or "if" or "then" or "else") {
Add(keyword.Value, value); }
            else if (keyword.Name is "allOf" or "anyOf" or "oneOf")

```

```

        { foreach (JsonElement child in keyword.Value.EnumerateArray()) {
Add(child, value); } }
        else if (keyword.Name is "enum" or "const") { cost +=
keyword.Value.GetRawText().Length * (long)value.GetRawText().Length; }
        else if (keyword.Name == "pattern" && value.ValueKind ==
JsonValueKind.String)
        { cost += keyword.Value.GetString().Length *
(long)value.GetString().Length; }
        else if (keyword.Name == "uniqueItems" && value.ValueKind ==
JsonValueKind.Array)
        { cost += value.GetArrayLength() * (long)value.GetRawText().Length; }
        if (cost > 2_000_000) { throw new ConfigurationValidationException("Schema
expansion exceeds the validation work limit."); }
    }
    memo[(node, value)] = cost;
    return cost;
}
}

```

DeploymentSettingsService.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.DTOs.Vendor.DeploymentSettings;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Coordinates deployment settings operations.
/// </summary>
public sealed class DeploymentSettingsService(TenantOperationFactory operations,
VendorArtifactArchive archive, VendorSecurityAudit audit)
{
    /// <summary>
    /// Deserializes stored deployment settings when a JSON value is present.
    /// </summary>
    public static T? Decode<T>(string? json) => json is null ? default :
JsonSerializer.Deserialize<T>(json);

    /// <summary>
    /// Reads deployment settings for an authorized product or version.
    /// </summary>
    public async Task<DeploymentSettingsResponse> GetAsync(TenantActor actor, Guid productId,
Guid? versionId, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.read", false, ct);
        Product product = await op.Database.Products.SingleOrDefaultAsync(x => x.Id ==
productId, ct) ?? throw new ApiAccessException(404);
        if (versionId is null) { return
new(Decode<DeploymentSettings>(product.DeploymentSettingsJson), null, false); }
        ProductVersion version = await op.Database.ProductVersions.SingleOrDefaultAsync(x
=> x.ProductId == productId && x.Id == versionId, ct) ?? throw new ApiAccessException(404);
        FileArtifact? installer = await op.Database.ProductVersionArtifacts.Where(x =>
x.ProductVersionId == versionId && x.ArtifactType == "Installer").Select(x =>
x.FileArtifact).SingleOrDefaultAsync(ct);
    }
}

```

```

        return new(Decode<DeploymentSettings>(version.DeploymentSettingsJson ??
(version.Status == "Draft" ? product.DeploymentSettingsJson : null)),
        Decode<InstallerIdentity>(installer?.InstallerIdentityJson), version.Status
== "Published" && version.DeploymentSettingsJson is not null);
    }

    /// <summary>
    /// Validates and saves deployment settings, including permitted initialization of
published versions.
    /// </summary>
    public async Task<DeploymentSettingsResponse> PutAsync(TenantActor actor, Guid productId,
Guid? versionId, DeploymentSettings settings, bool initialize, CancellationToken ct)
    {
        await using TenantOperation op = await operations.OpenAsync(actor,
"products.manage", true, ct);
        VendorDbContext db = op.Database;
        Product product = await db.Products.SingleOrDefaultAsync(x => x.Id == productId,
ct) ?? throw new ApiAccessException(404);
        string json = JsonSerializer.Serialize(settings);
        if (versionId is null)
        {
            if (product.DeploymentSettingsJson != json)
            {
                product.DeploymentSettingsJson = json;
                audit.Record(db, actor.UserId, "ProductDeploymentSettingsUpdated",
productId, entityType: "Product");
                await db.SaveChangesAsync(ct);
            }
            return new(settings, null, false);
        }
        ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x =>
x.ProductId == productId && x.Id == versionId, ct) ?? throw new ApiAccessException(404);
        if (initialize)
        {
            if (version.Status != "Published" || version.PublishedAt is null)
            { throw new ApiConflictException("published-version-required", "Initialize
applies only to an existing published release."); }
            if (version.DeploymentSettingsJson is not null &&
version.DeploymentSettingsJson != json)
            { throw new ApiConflictException("published-version-immutable", "Published
deployment settings cannot be changed."); }
        }
        else { BridgeControl.Repositories.Vendor.ProductRepository.RequireDraft(version); }

        FileArtifact? artifact = await db.ProductVersionArtifacts.Where(x =>
x.ProductVersionId == versionId && x.ArtifactType == "Installer").Select(x =>
x.FileArtifact).SingleOrDefaultAsync(ct);
        if (initialize && (artifact is null || artifact.Id !=
version.PublishedInstallerArtifactId))
        { throw new ApiConflictException("installer-required", "The original published
installer is required."); }
        bool changed = version.DeploymentSettingsJson != json;
        if (artifact is not null && artifact.InstallerIdentityJson is null)
        {
            await using Stream stream = await archive.OpenAsync(op.Tenant.ArchivePath,
artifact, ct);
            artifact.InstallerIdentityJson =
JsonSerializer.Serialize(MsiArtifactValidator.ReadIdentity(((FileStream) stream).Name));
            changed = true;
        }
        InstallerIdentity? identity =
Decode<InstallerIdentity>(artifact?.InstallerIdentityJson);

```



```

        if (initialize && (identity?.UpgradeCode is null || identity.UpgradeCode ==
Guid.Empty))
        { throw new ApiConflictException("upgrade-code-required", "The installer needs an
UpgradeCode for managed deployment."); }
        version.DeploymentSettingsJson = json;
        if (changed)
        {
            audit.Record(db, actor.UserId, initialize ?
"PublishedDeploymentSettingsInitialized" : "VersionDeploymentSettingsUpdated", version.Id,
entityType: "ProductVersion");
            await db.SaveChangesAsync(ct);
        }
        return new(settings, identity, version.Status == "Published");
    }
}

```

Draft202012SchemaArtifactValidator.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Validates Draft 2020-12 schemas using the supported keywords and resource limits.
/// </summary>
public sealed class Draft202012SchemaArtifactValidator : ISchemaArtifactValidator
{
    public const string DialectUri = "https://json-schema.org/draft/2020-12/schema";
    public string Dialect => DialectUri;
    private static readonly HashSet<string> Types = ["null", "boolean", "object", "array",
"number", "integer", "string"];

    /// <summary>
    /// Checks that a schema fits the supported, bounded Draft 2020-12 profile.
    /// </summary>
    public void Validate(JsonElement root, CancellationToken ct)
    {
        int values = 0; CheckJson(root, ref values);
        Require(root.ValueKind == JsonValueKind.Object && root.TryGetProperty("$schema",
out JsonElement dialect)
            && dialect.ValueKind == JsonValueKind.String && dialect.GetString() ==
Dialect, "#", "Declare the supported $schema dialect.");
        Dictionary<string, List<string>> graph = [];
        Visit(root, "#", 0, graph, ct);
        Dictionary<string, int> finished = []; HashSet<string> active = [];
        foreach (string node in graph.Keys) { CheckReferences(node, graph, finished,
active, 0); }
    }

    /// <summary>
    /// Checks schema JSON structure and counts nodes against the supported limits.
    /// </summary>
    private static void CheckJson(JsonElement value, ref int count)
    {
        Require(++count <= 16384, "#", "Schema exceeds the JSON value limit.");
        if (value.ValueKind == JsonValueKind.Object)
        {
            HashSet<string> names = [];
            foreach (JsonProperty property in value.EnumerateObject())

```

```

        {
            Require(names.Add(property.Name), "#", "Duplicate JSON property
names are not supported.");
            CheckJson(property.Value, ref count);
        }
    }
    else if (value.ValueKind == JsonValueKind.Array) { foreach (JsonElement item in
value.EnumerateArray()) { CheckJson(item, ref count); } }
    else if (value.ValueKind == JsonValueKind.Number)
    { Require(value.TryGetDouble(out double number) && double.IsFinite(number), "#",
"Numbers must be finite IEEE-754 values."); }
    }

    /// <summary>
    /// Visits schema keywords and builds the local reference graph while enforcing profile
limits.
    /// </summary>
    private static void Visit(JsonElement schema, string path, int depth, Dictionary<string,
List<string>> graph, CancellationToken ct)
    {
        ct.ThrowIfCancellationRequested();
        Require(depth <= 32 && graph.Count < 2048, path, "Schema exceeds the depth or
schema-node limit.");
        Require(schema.ValueKind is JsonValueKind.Object or JsonValueKind.True or
JsonValueKind.False, path, "Expected an object or boolean schema.");
        graph.Add(path, []);
        if (schema.ValueKind != JsonValueKind.Object) { return; }
        foreach (JsonProperty keyword in schema.EnumerateObject())
        {
            string at = path + "/" + Escape(keyword.Name); JsonElement value =
keyword.Value;

            switch (keyword.Name)
            {
                case "$schema": Require(path == "#" && value.ValueKind ==
JsonValueKind.String && value.GetString() == DialectUri, at, "Only the root dialect declaration
is supported."); break;
                case "$ref":
                    Require(value.ValueKind == JsonValueKind.String &&
value.GetString()!.StartsWith("#/", StringComparison.Ordinal)
                    && !value.GetString()!.Contains('%'), at, "Only
local JSON Pointer references are supported.");
                    graph[path].Add(value.GetString()!); break;
                case "$defs":
                case "properties":
                    Require(value.ValueKind == JsonValueKind.Object, at,
"Expected a map of schemas.");
                    foreach (JsonProperty child in value.EnumerateObject())
                    {
                        string childPath = at + "/" + Escape(child.Name);
                        // $defs are reusable declarations, not evaluation
edges until referenced.

                        // Properties descend into the instance and may
safely lead back to this schema.

                        Visit(child.Value, childPath, depth + 1, graph, ct);
                    }
                    break;
                case "additionalProperties":
                case "items":
                    Visit(value, at, depth + 1, graph, ct); break;
                case "not":
                case "if":
                case "then":
            }
        }
    }

```

```

        case "else":
            graph[path].Add(at); Visit(value, at, depth + 1, graph, ct);
break;

        case "allOf":
        case "anyOf":
        case "oneOf":
        case "prefixItems":
            Require(value.ValueKind == JsonValueKind.Array &&
value.GetArrayLength() is > 0 and <= 64, at, "Expected between 1 and 64 schemas.");
            int index = 0;
            foreach (JsonElement child in value.EnumerateArray())
            { string childPath = at + "/" + index++; if (keyword.Name !=
"prefixItems") { graph[path].Add(childPath); } Visit(child, childPath, depth + 1, graph, ct); }
            break;
        case "type":
            if (value.ValueKind == JsonValueKind.String) {
Require(Types.Contains(value.GetString()!), at, "Unknown type."); }
            else
            {
                Require(value.ValueKind == JsonValueKind.Array &&
value.GetArrayLength() is > 0 and <= 7, at, "Expected a type or nonempty type array.");
                HashSet<string> unique = [];
                foreach (JsonElement item in value.EnumerateArray())
                { Require(item.ValueKind == JsonValueKind.String &&
Types.Contains(item.GetString()!) && unique.Add(item.GetString()!), at, "Types must be known and
distinct."); }
            }
            break;
        case "required":
            Require(value.ValueKind == JsonValueKind.Array &&
value.GetArrayLength() <= 1024, at, "Expected a bounded string array.");
            HashSet<string> required = [];
            foreach (JsonElement item in value.EnumerateArray()) {
Require(item.ValueKind == JsonValueKind.String && required.Add(item.GetString()!), at, "Required
names must be distinct strings."); }
            break;
        case "enum":
            Require(value.ValueKind == JsonValueKind.Array &&
value.GetArrayLength() is > 0 and <= 512, at, "Expected between 1 and 512 enum values.");
            JsonElement[] choices = value.EnumerateArray().ToArray();
            for (int i = 0; i < choices.Length; i++) { for (int j = 0; j
< i; j++) { Require(!JsonElement.DeepEquals(choices[i], choices[j]), at, "Enum values must be
distinct."); } }
            break;
        case "minLength":
        case "maxLength":
        case "minItems":
        case "maxItems":
        case "minProperties":
        case "maxProperties":
            Require(value.ValueKind == JsonValueKind.Number &&
value.TryGetDecimal(out decimal bound) && bound >= 0 && decimal.Truncate(bound) == bound && bound
<= int.MaxValue, at, "Expected a nonnegative 32-bit integer."); break;
        case "minimum":
        case "maximum":
        case "exclusiveMinimum":
        case "exclusiveMaximum":
            Require(value.ValueKind == JsonValueKind.Number, at,
"Expected a number."); break;
        case "multipleOf": Require(value.ValueKind == JsonValueKind.Number
&& value.GetDouble() > 0, at, "Expected a positive number."); break;
        case "uniqueItems":

```

```

        case "readOnly":
        case "writeOnly":
        case "deprecated":
            Require(value.ValueKind is JsonValueKind.True or
JsonValueKind.False, at, "Expected a boolean."); break;
        case "pattern":
            Require(value.ValueKind == JsonValueKind.String, at,
"Expected a pattern string.");
            BoundedPatternKeyword.Create(value.GetString()); break;
        case "format":
            Require(value.ValueKind == JsonValueKind.String &&
value.GetString() == "uri", at, "Only the uri format is supported."); break;
        case "title":
        case "description":
        case "$comment":
            Require(value.ValueKind == JsonValueKind.String, at,
"Expected a string."); break;
        case "examples": Require(value.ValueKind == JsonValueKind.Array,
at, "Expected an array."); break;
        case "default": case "const": break;
        default: throw new ArtifactValidationException($"Unsupported schema
keyword at {SafePath(at)}.");
    }
}

/// <summary>
/// Rejects cyclic or excessively deep schema reference chains.
/// </summary>
private static int CheckReferences(string node, Dictionary<string, List<string>> graph,
Dictionary<string, int> finished, HashSet<string> active, int depth)
{
    Require(depth <= 64, node, "Reference expansion is too deep.");
    Require(graph.ContainsKey(node), node, "Reference must resolve to a schema in this
document.");
    if (finished.TryGetValue(node, out int cached))
    { Require(depth + cached <= 64, node, "Reference expansion is too deep."); return
cached; }
    Require(active.Add(node), node, "Reference cycles must descend into an object
property or array item.");
    int height = 0;
    foreach (string child in graph[node]) { height = Math.Max(height, 1 +
CheckReferences(child, graph, finished, active, depth + 1)); }
    Require(depth + height <= 64, node, "Reference expansion is too deep.");
    active.Remove(node); finished.Add(node, height); return height;
}

/// <summary>
/// Escapes a property name for use in a JSON pointer.
/// </summary>
private static string Escape(string value) => value.Replace("~", "~0").Replace("/", "~1");

/// <summary>
/// Truncates schema locations for bounded diagnostic messages.
/// </summary>
private static string SafePath(string value) => value.Length <= 180 ? value : value[..180]
+ "...";

/// <summary>
/// Raises a schema validation error when a profile requirement is not met.
/// </summary>
private static void Require(bool condition, string path, string message)

```

```

        { if (!condition) { throw new ArtifactValidationException($"{message} At
{SafePath(path)}"); } }
    }
}

```

Services/Vendor

EffectiveConfigurationGenerator.cs

```

using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>Trusted internal generation; HTTP clients have only read operations. Target switches
require an explicit caller decision.</summary>
public sealed class EffectiveConfigurationGenerator(PlatformDbContext platform, ITenantResolver
resolver,
    ConfigurationValues values, ConfigurationArtifactReader reader, VendorArtifactArchive
archive, VendorSecurityAudit audit, TimeProvider clock)
{
    /// <summary>
    /// Generates an effective configuration while holding the vendor operation lock.
    /// </summary>
    public async Task<Guid> GenerateAsync(Guid vendorId, Guid installationId, Guid versionId,
bool selectTarget = false, CancellationToken ct = default)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext tenant = await resolver.ResolveForAgentAsync(vendorId,
ct) ?? throw new ApiAccessException(404);
        return await GenerateLockedAsync(tenant, installationId, versionId, selectTarget,
ct);
    }

    /// <summary>
    /// Merges and validates configuration layers inside an existing locked tenant operation.
    /// </summary>
    internal async Task<Guid> GenerateLockedAsync(TenantContext tenant, Guid installationId,
Guid versionId, bool selectTarget, CancellationToken ct)
    {
        VendorDbContext db = tenant.Database;
        // Migration commits already own a serializable transaction covering profiles,
effective state and approvals.
        await using IDbContextTransaction? transaction = db.Database.CurrentTransaction is
null
            ? await
db.Database.BeginTransactionAsync(System.Data.IsolationLevel.Serializable, ct) : null;
        Installation installation = await db.Installations.Include(x =>
x.CurrentEffectiveConfigurationRevision)

```

```

        .SingleOrDefaultAsync(x => x.Id == installationId, ct) ?? throw new
ApiAccessException(404);
        ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x => x.Id
== versionId && x.ProductId == installation.ProductId, ct)
        ?? throw new ApiConflictException("configuration-version", "The target
version must belong to the installation's product.");
        if (version.Status is not ("Draft" or "Published")) { throw new
ApiConflictException("configuration-version", "The target version is unavailable."); }
        Guid? schemaId = version.Status == "Published" ? version.PublishedSchemaArtifactId
: await db.ProductVersionArtifacts.Where(x => x.ProductVersionId ==
versionId && x.ArtifactType == "Schema").Select(x =>
(Guid?)x.FileArtifactId).SingleOrDefaultAsync(ct);
        FileArtifact schemaArtifact = await db.FileArtifacts.SingleOrDefaultAsync(x =>
x.Id == schemaId && x.ArtifactType == "Schema", ct)
        ?? throw new ApiConflictException("schema-required", "The target version
needs a valid schema.");
        JsonElement schema = await reader.ReadAsync(tenant.ArchivePath, schemaArtifact,
true, ct);
        List<ConfigurationProfile> profiles = await db.ConfigurationProfiles.Where(x =>
(x.ProductId == installation.ProductId && x.InstallationId == null
&& (x.CustomerId == null || x.CustomerId == installation.CustomerId)
|| (x.InstallationId == installation.Id && x.ProductId == null &&
x.CustomerId == null)).ToListAsync(ct);
        Guid?[] sourceIds = new Guid?[3];
        ConfigurationRevision?[] sources = new ConfigurationRevision?[3];
        for (int layer = 0; layer < 3; layer++)
        {
            ConfigurationProfile? profile = profiles.SingleOrDefault(x => layer == 0 ?
x.ProductId != null && x.CustomerId == null
: layer == 1 ? x.CustomerId == installation.CustomerId :
x.InstallationId == installation.Id);
            if (layer == 0 && version.Status == "Published")
            {
                if (version.PublishedDefaultRevisionId is Guid pinned)
                {
                    sources[layer] = await db.ConfigurationRevisions.Include(x =>
x.FileArtifact).SingleOrDefaultAsync(x => x.Id == pinned, ct);
                    if (sources[layer] is null || profile is null ||
sources[layer]!.ConfigurationProfileId != profile.Id || sources[layer]!.ProductVersionId !=
versionId)
                    { throw new ApiConflictException("configuration-source",
"The published default has invalid source ownership."); }
                }
                else if (profile is not null)
                {
                    sources[layer] = await db.ConfigurationRevisions.Include(x =>
x.FileArtifact).Where(x => x.ConfigurationProfileId == profile.Id && x.ProductVersionId ==
versionId)
                    .OrderByDescending(x =>
x.RevisionNumber).FirstOrDefaultAsync(ct);
                }
                sourceIds[layer] = sources[layer]?.Id;
            }
            List<JsonElement> layers = [];
            foreach (ConfigurationRevision? source in sources)
            {
                if (source is null) { continue; }
                if (source.FileArtifact.ArtifactType != "Configuration") { throw new
ApiConflictException("configuration-source", "A source configuration artifact is invalid."); }
                JsonElement input = await reader.ReadAsync(tenant.ArchivePath,
source.FileArtifact, false, ct);

```

```

        values.Validate(schema, input, ct); layers.Add(input);
    }
    byte[] canonical = values.Merge(layers.ToArray());
    using JsonDocument merged = JsonDocument.Parse(canonical); values.Validate(schema,
merged.RootElement, ct);
    string fingerprint =
Convert.ToHexStringLower(SHA256.HashData(Encoding.UTF8.GetBytes(
    $"{version.Id:N}|{schemaArtifact.Id:N}|{string.Join('|', sourceIds.Select(x
=> x?.ToString("N") ?? "absent"))}|{Convert.ToHexStringLower(SHA256.HashData(canonical))}"));
    EffectiveConfigurationRevision? revision = await
db.EffectiveConfigurationRevisions.Include(x => x.FileArtifact)
    .SingleOrDefaultAsync(x => x.InstallationId == installation.Id &&
x.InputFingerprint == fingerprint, ct);
    bool publishCurrent = selectTarget ||
installation.CurrentEffectiveConfigurationRevision is null
    || installation.CurrentEffectiveConfigurationRevision.ProductVersionId ==
versionId;
    if (revision is not null)
    {
        await reader.ReadAsync(tenant.ArchivePath, revision.FileArtifact, false,
ct);
        if (publishCurrent && installation.CurrentEffectiveConfigurationRevisionId
!= revision.Id)
        {
            installation.CurrentEffectiveConfigurationRevisionId = revision.Id;
            audit.Record(db, Guid.Empty, "EffectiveConfigurationSelected",
installation.Id, entityType: "Installation", actorType: "System", resourceId: revision.Id);
            await db.SaveChangesAsync(ct);
        }
        if (transaction is not null) { await transaction.CommitAsync(ct); }
        return revision.Id;
    }
    // Together with the serializable snapshot and lifecycle lock, rowversions reject
stale external writers.
    db.Entry(version).Property(x => x.Version).IsModified = true;
    foreach (ConfigurationProfile? profile in profiles) { db.Entry(profile).Property(x
=> x.LastRevisionNumber).IsModified = true; }
    installation.LastEffectiveRevisionNumber =
checked(installation.LastEffectiveRevisionNumber + 1);
    await using StagedArtifact staged = await
archive.StageConfigurationAsync(tenant.ArchivePath, canonical, ct);
    revision = new()
    {
        Id = Guid.NewGuid(),
        InstallationId = installation.Id,
        ProductId = installation.ProductId,
        ProductVersionId = versionId,
        SchemaArtifactId = schemaArtifact.Id,
        InputFingerprint = fingerprint,
        RevisionNumber = installation.LastEffectiveRevisionNumber,
        CreatedAt = clock.UtcNow(),
        FileArtifact = new FileArtifact
        {
            Id = Guid.NewGuid(),
            ArtifactType = "Configuration",
            FileName = staged.FileName,
            ContentType = staged.ContentType,
            StorageIdentifier = staged.StorageIdentifier,
            Sha256 = staged.Sha256,
            SizeBytes = staged.SizeBytes,
            CreatedAt = clock.UtcNow()
        }
    }

```

```

        };
        foreach (Guid sourceId in sourceIds.OfType<Guid>()) {
            revision.SourceRevisions.Add(new() { ConfigurationRevisionId = sourceId });
            db.EffectiveConfigurationRevisions.Add(revision);
            staged.Promote();
            // Insert history before setting the circular installation -> effective revision
            pointer.

            await db.SaveChangesAsync(ct);
            if (publishCurrent) { installation.CurrentEffectiveConfigurationRevisionId =
revision.Id; }
            audit.Record(db, Guid.Empty, "EffectiveConfigurationGenerated", installation.Id,
entityType: "Installation", actorType: "System", resourceId: revision.Id);
            await db.SaveChangesAsync(ct); if (transaction is not null) { await
transaction.CommitAsync(ct); }
            return revision.Id;
        }
    }
}

```

EffectiveConfigurationProcessor.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>Bounded, restart-safe fanout and recalculation. Every invocation uses current
sources, never a cached calculation.</summary>
public sealed class EffectiveConfigurationProcessor(PlatformDbContext platform, ITenantResolver
resolver,
    EffectiveConfigurationGenerator generator, VendorSecurityAudit audit, TimeProvider clock)
: IVendorProcessor
{
    /// <summary>
    /// Expands configuration changes and processes queued generation work for one vendor.
    /// </summary>
    public async Task ProcessVendorAsync(Guid vendorId, CancellationToken ct = default)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await resolver.ResolveForAgentAsync(vendorId,
ct);

        if (tenant is null) { return; }
        VendorDbContext db = tenant.Database;
        await ExpandSourceAsync(db, ct); await ExpandSignalAsync(db, ct);
        DateTimeOffset now = clock.GetUtcNow();
        Guid[] pending = await db.EffectiveConfigurationWork.Where(x => x.Status ==
"Pending" && x.NextAttemptAt <= now)
            .OrderBy(x => x.NextAttemptAt).ThenBy(x => x.Id).Select(x =>
x.Id).Take(5).ToArrayAsync(ct);
        foreach (Guid id in pending)
        {
            db.ChangeTracker.Clear();
            EffectiveConfigurationWork work = await
db.EffectiveConfigurationWork.SingleAsync(x => x.Id == id, ct);
            try
            {

```



```

        await generator.GenerateLockedAsync(tenant, work.InstallationId,
work.ProductVersionId, false, ct);
        work.Status = "Succeeded"; work.CompletedGeneration =
work.RequestedGeneration; work.LastError = null;
        work.UpdatedAt = clock.GetUtcNow(); await db.SaveChangesAsync(ct);
    }
    catch (OperationCanceledException) when (ct.IsCancellationRequested) {
throw; }

    catch (Exception error)
    {
        // Generation has rolled back before this new outcome transaction;
a prior valid pointer survives.
        db.ChangeTracker.Clear(); work = await
db.EffectiveConfigurationWork.SingleAsync(x => x.Id == id, ct);
        work.Attempts++; work.UpdatedAt = clock.GetUtcNow();
        bool validation = error is ConfigurationValidationException or
ApiConflictException or ApiAccessErrorException;
        work.Status = validation || work.Attempts >= 3 ? "Failed" :
"Pending";
        work.NextAttemptAt = clock.GetUtcNow().AddSeconds(30 *
work.Attempts);
        string message = error is ConfigurationValidationException ?
error.Message
            : error is ApiConflictException conflict ? conflict.Code :
error is ApiAccessErrorException ? "SourceUnavailable" : "GenerationFailed:" + error.GetType().Name;
        work.LastError = message[..Math.Min(2000, message.Length)];
        audit.Record(db, Guid.Empty, "EffectiveConfigurationFailed",
work.InstallationId, work.Status, "Installation", "System", work.Id);
        await db.SaveChangesAsync(ct);
    }
}

}

/// <summary>
/// Fans out a source revision change to affected installation and version targets.
/// </summary>
private async Task ExpandSourceAsync(VendorDbContext db, CancellationToken ct)
{
    ConfigurationSourceChange? change = await db.ConfigurationSourceChanges.Where(x =>
x.ProcessedAt == null).OrderBy(x => x.CreatedAt).ThenBy(x => x.Id).FirstOrDefaultAsync(ct);
    if (change is null) { return; }
    ConfigurationRevision source = await db.ConfigurationRevisions.Include(x =>
x.ConfigurationProfile).Include(x => x.ProductVersion).SingleAsync(x => x.Id ==
change.ConfigurationRevisionId, ct);
    ConfigurationProfile profile = source.ConfigurationProfile;
    IQueryable<Installation> query = db.Installations.Where(x => x.ProductId ==
source.ProductVersion.ProductId);
    if (profile.InstallationId is Guid installation) { query = query.Where(x => x.Id
== installation); }
    else
    {
        query = query.Where(x => x.ProductId == profile.ProductId);
        if (profile.CustomerId is Guid customer) { query = query.Where(x =>
x.CustomerId == customer); }
    }
    if (change.LastInstallationId is Guid cursor) { query = query.Where(x =>
x.Id.CompareTo(cursor) > 0); }
    Guid[] targets = await query.OrderBy(x => x.Id).Select(x =>
x.Id).Take(100).ToArrayAsync(ct);
    await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);

```

```

        foreach (Guid target in targets) { await EnqueueAsync(db, target,
source.ProductVersionId, ct); }
        if (targets.Length > 0) { change.LastInstallationId = targets[^1]; }
        if (targets.Length < 100) { change.ProcessedAt = clock.GetUtcNow(); }
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
    }

    /// <summary>
    /// Expands a configuration signal into installation generation work.
    /// </summary>
    private async Task ExpandSignalAsync(VendorDbContext db, CancellationToken ct)
    {
        EffectiveConfigurationSignal? signal = await
db.EffectiveConfigurationSignals.Where(x => x.ProcessedAt == null).OrderBy(x =>
x.CreatedAt).ThenBy(x => x.Id).FirstOrDefaultAsync(ct);
        if (signal is null) { return; }
        IQueryable<Guid> targets;
        if (signal.ProductVersionId is Guid version)
        {
            Guid product = await db.ProductVersions.Where(x => x.Id ==
version).Select(x => x.ProductId).SingleAsync(ct);
            targets = db.Installations.Where(x => x.ProductId == product).Select(x =>
x.Id);
        }
        else
        {
            Guid product = await db.Installations.Where(x => x.Id ==
signal.InstallationId).Select(x => x.ProductId).SingleAsync(ct);
            targets = db.ProductVersions.Where(x => x.ProductId == product &&
(x.PublishedSchemaArtifactId != null || x.Artifacts.Any(a => a.ArtifactType ==
"Schema"))).Select(x => x.Id);
        }
        if (signal.LastTargetId is Guid cursor) { targets = targets.Where(x =>
x.CompareTo(cursor) > 0); }
        Guid[] ids = await targets.OrderBy(x => x).Take(100).ToArrayAsync(ct);
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        foreach (Guid id in ids) { await EnqueueAsync(db, signal.InstallationId ?? id,
signal.ProductVersionId ?? id, ct); }
        if (ids.Length > 0) { signal.LastTargetId = ids[^1]; }
        if (ids.Length < 100) { signal.ProcessedAt = clock.GetUtcNow(); }
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
    }

    /// <summary>
    /// Schedules effective-configuration generation for an installation and version.
    /// </summary>
    private async Task EnqueueAsync(VendorDbContext db, Guid installationId, Guid versionId,
CancellationToken ct)
    {
        EffectiveConfigurationWork? work =
db.EffectiveConfigurationWork.Local.SingleOrDefault(x => x.InstallationId == installationId &&
x.ProductVersionId == versionId)
        ?? await db.EffectiveConfigurationWork.SingleOrDefaultAsync(x =>
x.InstallationId == installationId && x.ProductVersionId == versionId, ct);
        if (work is null) { work = new() { InstallationId = installationId,
ProductVersionId = versionId }; db.EffectiveConfigurationWork.Add(work); }
        else { work.RequestedGeneration = checked(work.RequestedGeneration + 1); }
        work.Status = "Pending"; work.Attempts = 0; work.LastError = null;
work.NextAttemptAt = work.UpdatedAt = clock.GetUtcNow();
    }
}

```

EffectiveConfigurationWorker.cs

```
using BridgeControl.Options;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>Runs recurring effective configuration work until the host stops.</summary>
public sealed class EffectiveConfigurationWorker(IServiceScopeFactory scopes,
IOptions<EffectiveConfigurationOptions> configured,
    ILogger<EffectiveConfigurationWorker> logger) :
VendorWorker<EffectiveConfigurationProcessor>(
    scopes, () => configured.Value.WorkerEnabled, () => configured.Value.PollSeconds,
logger, "Effective configuration");
```

HistoryRetentionService.cs

```
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Models.Shared;
using BridgeControl.Options;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;
namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents cleanup result data.
/// </summary>
public sealed record CleanupResult(int Logs, int Heartbeats, int Credentials, int RefreshTokens,
int Sessions);

/// <summary>
/// Coordinates history retention operations.
/// </summary>
public sealed class HistoryRetentionService(PlatformDbContext platform, IVendorInfrastructure
infrastructure, IOOptions<OperationalHistoryOptions> options, TimeProvider clock)
{
    /// <summary>
    /// Opens the vendor database for history maintenance, including inactive vendors.
    /// </summary>
    public async Task<VendorDbContext> OpenVendorAsync(Guid vendorId, CancellationToken ct)
    {
        SoftwareVendor? vendor = await platform.SoftwareVendors.AsNoTracking().Include(x
=> x.Database).SingleOrDefaultAsync(x => x.Id == vendorId, ct);
        if (vendor?.Database?.Status != "Ready") { throw new ApiAccessException(404); }
        return infrastructure.OpenDatabase(vendor);
    }

    /// <summary>
    /// Deletes a bounded batch of expired vendor history while preserving required evidence.
    /// </summary>
    public async Task<CleanupResult> VendorAsync(Guid vendorId, CancellationToken ct =
default)
```

```

        {
            await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
            await using VendorDbContext db = await OpenVendorAsync(vendorId, ct);
            await using IDbContextTransaction tx = await
db.Database.BeginTransactionAsync(ct);
            OperationalHistoryOptions o = options.Value; DateTimeOffset now =
clock.GetUtcNow(); DateTimeOffset cutoff = now.AddDays(-o.CredentialGraceDays); int batch =
o.CleanupBatchSize;
            int logs = await db.OperationalLogs.Where(x => x.ReceivedAt <
now.AddDays(-o.LogDays)).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            int beats = await db.AgentHeartbeats.Where(x => x.ReceivedAt <
now.AddDays(-o.HeartbeatDays)).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            int credentials = await db.AgentCredentials.Where(x => x.ExpiresAt <
cutoff).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            credentials += await db.AgentEnrollments.Where(x => x.ExpiresAt <
cutoff).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            credentials += await db.PasswordResets.Where(x => x.ExpiresAt < cutoff).OrderBy(x
=> x.Id).Take(batch).ExecuteDeleteAsync(ct);
            credentials += await db.AccountActivations.Where(x => x.ExpiresAt <
cutoff).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            (int Tokens, int Sessions) auth = await AuthenticationAsync(db, cutoff, batch,
ct);
            await tx.CommitAsync(ct); return new(logs, beats, credentials, auth.Tokens,
auth.Sessions);
        }

        /// <summary>
        /// Deletes a bounded batch of expired platform history and authentication records.
        /// </summary>
        public async Task<CleanupResult> PlatformAsync(CancellationTokens ct = default)
        {
            await using PlatformVendorLock gate = await
PlatformVendorLock.AcquireAsync(platform, ct);
            await using IDbContextTransaction tx = await
platform.Database.BeginTransactionAsync(ct);
            (int Tokens, int Sessions) auth = await AuthenticationAsync(platform,
clock.GetUtcNow().AddDays(-options.Value.CredentialGraceDays), options.Value.CleanupBatchSize,
ct);
            await tx.CommitAsync(ct); return new(0, 0, 0, auth.Tokens, auth.Sessions);
        }

        /// <summary>
        /// Removes expired authentication data while preserving replay evidence for live
sessions.
        /// </summary>
        private static async Task<(int Tokens, int Sessions)> AuthenticationAsync(DbContext db,
DateTimeOffset cutoff, int batch, CancellationTokens ct)
        {
            // Never prune a consumed token while its owning session can still detect reuse.
            int tokens = await db.Set<RefreshToken>().Where(x => x.ExpiresAt < cutoff &&
x.AuthenticationSession.ExpiresAt < cutoff).OrderBy(x =>
x.Id).Take(batch).ExecuteDeleteAsync(ct);
            int sessions = await db.Set<AuthenticationSession>().Where(x => x.ExpiresAt <
cutoff && !x.RefreshTokens.Any()).OrderBy(x => x.Id).Take(batch).ExecuteDeleteAsync(ct);
            return (tokens, sessions);
        }
    }
}

```

HistoryRetentionWorker.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Platform;
using BridgeControl.Options;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;
namespace BridgeControl.Services.Vendor;

/// <summary>
/// Runs recurring history retention work until the host stops.
/// </summary>
public sealed class HistoryRetentionWorker(IServiceScopeFactory scopes,
IOptions<OperationalHistoryOptions> options, ILogger<HistoryRetentionWorker> logger) :
BackgroundService
{
    /// <summary>
    /// Runs periodic bounded history cleanup until the host stops.
    /// </summary>
    protected override async Task ExecuteAsync(CancellationToken ct)
    {
        if (!options.Value.WorkerEnabled) { return; }
        Guid? cursor = null;
        while (!ct.IsCancellationRequested)
        {
            try
            {
                await Task.Delay(TimeSpan.FromSeconds(options.Value.PollSeconds),
ct);

                await using AsyncServiceScope scope = scopes.CreateAsyncScope();
                await
scope.ServiceProvider.GetRequiredService<HistoryRetentionService>().PlatformAsync(ct);
                PlatformDbContext db =
scope.ServiceProvider.GetRequiredService<PlatformDbContext>();
                IQueryable<SoftwareVendor> query = db.SoftwareVendors.Where(x =>
x.Database != null && x.Database.Status == "Ready");
                if (cursor is Guid last) { query = query.Where(x =>
x.Id.CompareTo(last) > 0); }
                Guid[] ids = await query.OrderBy(x => x.Id).Select(x =>
x.Id).Take(20).ToArrayAsync(ct); cursor = ids.Length < 20 ? null : ids[^1];
                foreach (Guid id in ids)
                {
                    try
                    {
                        await using AsyncServiceScope vendorScope =
scopes.CreateAsyncScope();

                        CleanupResult result = await
vendorScope.ServiceProvider.GetRequiredService<HistoryRetentionService>().VendorAsync(id, ct);
                        if (result != new CleanupResult(0, 0, 0, 0, 0)) {
logger.LogInformation("History cleanup completed for vendor {VendorId}: {Counts}.", id, result);
}

                    }
                    catch (OperationCanceledException) when
(ct.IsCancellationRequested) { throw; }
                    catch (Exception e) { logger.LogWarning("History cleanup
failed for vendor {VendorId}: {ErrorType}.", id, e.GetType().Name); }
                }
            }
            catch (OperationCanceledException) when (ct.IsCancellationRequested) {
break; }
        }
    }
}
```

```

        catch (Exception e) { logger.LogWarning("History cleanup discovery failed:
{ErrorType}.", e.GetType().Name); }
    }
}

```

IIInvitationDelivery.cs

```

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Delivers one invitation without exposing its credential through the public API.
/// </summary>
public interface IInvitationDelivery
{
    /// <summary>
    /// Delivers an account invitation with its credential and expiry time.
    /// </summary>
    Task DeliverAsync(Guid invitationId, string email, string token, DateTimeOffset expiresAt,
CancellationTokens cancellationTokens);
}

```

InstallationStatePolicy.cs

```

using System.Text.Json;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents state target data.
/// </summary>
public sealed record StateTarget(ProductVersion Version, EffectiveConfigurationRevision
Configuration, FileArtifact Installer);

/// <summary>
/// Validates desired targets, artifact delivery, and reconciliation scheduling.
/// </summary>
public sealed class InstallationStatePolicy(LicenseEligibility licensing,
ConfigurationArtifactReader reader, ConfigurationValues values, VendorArtifactArchive archive)
{
    /// <summary>
    /// Rejects a desired-state change when the installation is ineligible for deployment.
    /// </summary>
    public async Task RequireEligibleAsync(VendorDbContext db, Guid installation,
CancellationTokens ct)
    {
        DeploymentEligibility eligibility = await licensing.EvaluateAsync(db,
installation, ct);
        if (!eligibility.Allowed) { throw new ApiConflictException("state-ineligible",
eligibility.Reason); }
    }

    /// <summary>

```

```

    /// Resolves and validates the published version and effective configuration for a target.
    /// </summary>
    public async Task<StateTarget> TargetAsync(TenantContext tenant, Installation
installation, Guid versionId, Guid configurationId, CancellationToken ct)
    {
        VendorDbContext db = tenant.Database;
        ProductVersion version = await db.ProductVersions.SingleOrDefaultAsync(x => x.Id
== versionId && x.ProductId == installation.ProductId, ct) ?? throw new ApiAccessException(404);
        if (version.Status != "Published" || version.PublishedSchemaArtifactId is null ||
version.PublishedInstallerArtifactId is null)
        { throw new ApiConflictException("state-release-required", "Desired state requires
a published release."); }
        EffectiveConfigurationRevision configuration = await
db.EffectiveConfigurationRevisions.Include(x => x.FileArtifact).SingleOrDefaultAsync(x => x.Id ==
configurationId && x.InstallationId == installation.Id && x.ProductVersionId == versionId, ct)
        ?? throw new ApiAccessException(404);
        if (configuration.SchemaArtifactId != version.PublishedSchemaArtifactId)
        { throw new ApiConflictException("configuration-migration-required", "The
configuration does not match the release schema."); }
        FileArtifact schema = await db.FileArtifacts.SingleAsync(x => x.Id ==
version.PublishedSchemaArtifactId, ct);
        FileArtifact installer = await db.FileArtifacts.SingleAsync(x => x.Id ==
version.PublishedInstallerArtifactId, ct);
        if (schema.ArtifactType != "Schema" || installer.ArtifactType != "Installer" ||
configuration.FileArtifact.ArtifactType != "Configuration")
        { throw new ApiConflictException("state-artifact-invalid", "The release artifacts
are invalid."); }
        JsonElement schemaValues = await reader.ReadAsync(tenant.ArchivePath, schema,
true, ct);
        JsonElement configValues = await reader.ReadAsync(tenant.ArchivePath,
configuration.FileArtifact, false, ct); values.Validate(schemaValues, configValues, ct);
        await using Stream content = await OpenAsync(tenant.ArchivePath, installer, ct);
        return new(version, configuration, installer);
    }

    /// <summary>
    /// Opens a verified artifact from the vendor archive.
    /// </summary>
    public async Task<Stream> OpenAsync(string root, FileArtifact artifact, CancellationToken
ct)
    {
        try { return await archive.OpenAsync(root, artifact, ct); }
        catch (Exception error) when (error is IOException or InvalidOperationException or
UnauthorizedAccessException)
        { throw new ApiConflictException("state-artifact-unavailable", "The requested
artifact is unavailable or failed its integrity check."); }
    }

    /// <summary>
    /// Schedules installation-state reconciliation after new desired or reported evidence.
    /// </summary>
    public static async Task EnqueueAsync(VendorDbContext db, Guid installation,
DateTimeOffset now, CancellationToken ct)
    {
        StateReconciliationWork? work = await
db.StateReconciliationWork.SingleOrDefaultAsync(x => x.InstallationId == installation, ct);
        if (work is null) { work = new() { InstallationId = installation };
db.StateReconciliationWork.Add(work); }
        work.RequestedGeneration = checked(work.RequestedGeneration + 1); work.Status =
"Pending"; work.Attempts = 0; work.Reason = null; work.NextAttemptAt = work.UpdatedAt = now;
    }
}

```

ISchemaArtifactValidator.cs

```
using System.Text.Json;

namespace BridgeControl.Services.Vendor;

/// <summary>Validates schema documents for one declared JSON Schema dialect.</summary>
public interface ISchemaArtifactValidator
{
    string Dialect { get; }
    /// <summary>
    /// Validates a schema against the supported artifact profile.
    /// </summary>
    void Validate(JsonElement schema, CancellationToken ct);
}
```

IVendorProcessor.cs

```
namespace BridgeControl.Services.Vendor;

/// <summary>Processes a bounded batch of queued work for one vendor.</summary>
public interface IVendorProcessor
{
    Task ProcessVendorAsync(Guid vendorId, CancellationToken ct = default);
}
```

LicenseEligibility.cs

```
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents deployment eligibility data.
/// </summary>
public sealed record DeploymentEligibility(bool Allowed, string Reason);

/// <summary>Internal gate for future desired-state creation and work delivery; caller resolves
an active vendor and authorizes installation ownership.</summary>
public sealed class LicenseEligibility(TimeProvider clock)
{
    /// <summary>
    /// Checks active status and the inclusive start and exclusive end of a license's validity
    period.
    /// </summary>
    public static bool ValidAt(License license, DateTimeOffset now) => license.Status ==
"Active"
        && license.ValidFrom <= now && (license.ValidUntil is null || now <
license.ValidUntil);

    /// <summary>
```



```

        /// Counts active installations assigned to a license, optionally excluding one
        installation.
        /// </summary>
        public Task<int> ActiveCountAsync(VendorDbContext db, Guid licenseId, Guid?
exceptInstallation, CancellationToken ct) =>
            db.InstallationLicenseAssignments.CountAsync(x => x.LicenseId == licenseId &&
x.RemovedAt == null
                && x.Installation.Status == "Active" && (exceptInstallation == null ||
x.InstallationId != exceptInstallation), ct);

        /// <summary>
        /// Rejects license assignments with mismatched ownership, invalid dates, or exhausted
        capacity.
        /// </summary>
        public async Task RequireAssignmentAsync(VendorDbContext db, Installation installation,
License license, CancellationToken ct)
        {
            if (license.CustomerId != installation.CustomerId || license.ProductId !=
installation.ProductId)
            { throw new ApiConflictException("license-mismatch", "The license must match the
installation's customer and product."); }
            if (!ValidAt(license, clock.GetUtcNow())) { throw new
ApiConflictException("license-unavailable", "The license is disabled or outside its validity
period."); }
            if (license.MaxInstallations is int maximum && await ActiveCountAsync(db,
license.Id, installation.Id, ct) >= maximum)
            { throw new ApiConflictException("license-capacity", "The license has no available
active installation capacity."); }
        }

        /// <summary>
        /// Checks installation, customer, product, agent, and license eligibility for deployment.
        /// </summary>
        public async Task<DeploymentEligibility> EvaluateAsync(VendorDbContext db, Guid
installationId, CancellationToken ct = default)
        {
            Installation? installation = await db.Installations.AsNoTracking().Include(x =>
x.Customer).Include(x => x.Product)
                .SingleOrDefaultAsync(x => x.Id == installationId, ct);
            if (installation is null || installation.Status != "Active" ||
installation.Customer.Status != "Active" || installation.Product.Status != "Active")
            { return new(false, "InstallationUnavailable"); }
            if (installation.AgentId is not Guid agent || !await db.Agents.AnyAsync(x => x.Id
== agent && x.CustomerId == installation.CustomerId && x.Status == "Active", ct))
            { return new(false, "AgentUnavailable"); }
            License? license = await db.InstallationLicenseAssignments.Where(x =>
x.InstallationId == installationId && x.RemovedAt == null)
                .Select(x => x.License).AsNoTracking().SingleOrDefaultAsync(ct);
            if (license is null) { return new(false, "LicenseMissing"); }
            if (license.CustomerId != installation.CustomerId || license.ProductId !=
installation.ProductId || !ValidAt(license, clock.GetUtcNow()))
            { return new(false, "LicenseUnavailable"); }
            if (license.MaxInstallations is int maximum && await ActiveCountAsync(db,
license.Id, null, ct) > maximum) { return new(false, "LicenseCapacity"); }
            return new(true, "Eligible");
        }
    }
}

```

LocalInvitationDelivery.cs

```

using System.Net;
using System.Text;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Atomically writes development invitations as private local email files, with one file per
/// invitation.
/// </summary>
public class LocalInvitationDelivery(IOptions<VendorProvisioningOptions> options) :
    IInvitationDelivery
{
    /// <summary>
    /// Writes an account activation invitation to the protected local outbox.
    /// </summary>
    public Task DeliverAsync(Guid invitationId, string email, string token, DateTimeOffset
expiresAt, CancellationToken cancellationToken) => WriteAsync(invitationId, email, token,
expiresAt, false, cancellationToken);

    /// <summary>
    /// Writes a password-reset invitation to the protected local outbox.
    /// </summary>
    public Task DeliverResetAsync(Guid id, string email, string token, DateTimeOffset
expiresAt, CancellationToken ct) => WriteAsync(id, email, token, expiresAt, true, ct);

    /// <summary>
    /// Persists the invitation payload in the protected local delivery directory.
    /// </summary>
    private async Task WriteAsync(Guid invitationId, string email, string token,
DateTimeOffset expiresAt, bool reset, CancellationToken cancellationToken)
    {
        if (email.Contains('\r') || email.Contains('\n'))
        {
            throw new ArgumentException("The invitation email address is invalid.");
        }
        string directory = PrivateDirectory.Ensure(options.Value.InvitationDirectory);
        string target = Path.Combine(directory, $"{invitationId:N}.eml");
        if (File.Exists(target))
        {
            if (File.GetAttributes(target).HasFlag(FileAttributes.ReparsePoint))
            {
                throw new IOException("An invitation file cannot be a symbolic
link.");
            }
            return;
        }
        string temporary = Path.Combine(directory,
$"{invitationId:N}.{Guid.NewGuid():N}.tmp");
        string subject = reset ? "Reset your BridgeControl password" : "Activate your
BridgeControl account";
        string introduction = reset ? "A password reset was requested for your account." :
"You have been invited to a BridgeControl account.";
        string route = reset ? "/reset" : "/activate";
        string link =
$"{options.Value.FrontendBaseUrl.TrimEnd('/')}{route}?token={Uri.EscapeDataString(token)}";
        string label = reset ? "Reset token" : "Activation token";
        string instructions = "Open the link and choose your own password (10-128
characters). The token is filled in automatically.";
        string privacy = "Keep this email private. The link and token can be used once.";
    }
}

```

```

        string plainText =
        $"{introduction}\r\n\r\n{instructions}\r\n{link}\r\n\r\n{label}: {token}\r\nExpires at:
        {expiresAt:O}\r\n\r\n{privacy}\r\n";
        string html = $"<!doctype
html><html><body><p>{introduction}</p><p>{instructions}</p><p><a
href="{WebUtility.HtmlEncode(link)}">{subject}</a></p><p>{label}:
{WebUtility.HtmlEncode(token)}<br>Expires at:
{expiresAt:O}</p><p>{privacy}</p></body></html>\r\n";
        string boundary = $"bridgecontrol_{Guid.NewGuid():N}";
        string message = $"From: BridgeControl <noreply@bridgecontrol.local>\r\nTo:
{email}\r\nSubject: {subject}\r\nMIME-Version: 1.0\r\nContent-Type: multipart/alternative;
boundary="{boundary}"\r\n\r\n"
            + $"--{boundary}\r\nContent-Type: text/plain;
charset=utf-8\r\nContent-Transfer-Encoding: 8bit\r\n\r\n{plainText}\r\n"
            + $"--{boundary}\r\nContent-Type: text/html;
charset=utf-8\r\nContent-Transfer-Encoding: 8bit\r\n\r\n{html}\r\n--{boundary}--\r\n";
        try
        {
            await using (FileStream stream = new(temporary, FileMode.CreateNew,
            FileAccess.Write, FileShare.None, 4096, useAsync: true))
            {
                await stream.WriteAsync(Encoding.UTF8.GetBytes(message),
            cancellationToken);

                await stream.FlushAsync(cancellationToken);
            }
            File.Move(temporary, target);
        }
        finally
        {
            if (File.Exists(temporary)) { File.Delete(temporary); }
        }
    }
}

```

MsiArtifactValidator.cs

```

using System.Runtime.InteropServices;
using System.Text;
using BridgeControl.Api.Exceptions;
using BridgeControl.DTOS.Vendor.DeploymentSettings;

namespace BridgeControl.Services.Vendor;

/// <summary>Inspects a Windows Installer database in read-only mode; never runs installation
actions.</summary>
public static class MsiArtifactValidator
{
    /// <summary>
    /// Reads the MSI product code, version, and upgrade identity.
    /// </summary>
    public static InstallerIdentity ReadIdentity(string path)
    {
        uint database = 0;
        try
        {
            Check(MsiOpenDatabase(path, IntPtr.Zero, out database));
            Guid? upgrade = null;
            try { if (Guid.TryParse(Property(database, "UpgradeCode"), out Guid value))
            { upgrade = value; } }

```

```

        catch (ArtifactValidationException) { /* Older uploaded packages may have
no upgrade family. */ }
        return new(Guid.Parse(Property(database, "ProductCode")), upgrade,
Property(database, "ProductVersion"));
    }
    finally { if (database != 0) { MsiCloseHandle(database); } }
}

/// <summary>
/// Checks the package identity and required Windows Installer tables.
/// </summary>
public static void Validate(string path)
{
    if (!OperatingSystem.IsWindows()) { throw new
ArtifactValidationException("Installer validation requires the Windows deployment host."); }
    uint database = 0, summary = 0;
    try
    {
        Check(MsiOpenDatabase(path, IntPtr.Zero, out database));
        Check(MsiGetSummaryInformation(database, null, 0, out summary));
        uint length = 256;
        StringBuilder revision = new(256);
        Check(MsiSummaryInfoGetProperty(summary, 9, out uint type, out _, out _,
revision, ref length));
        if (type != 30 || !Guid.TryParseExact(revision.ToString(), "B", out _) {
Invalid(); }

        foreach (string name in new[] { "ProductCode", "ProductName",
"ProductVersion", "Manufacturer", "ProductLanguage" })
        {
            string value = Property(database, name);
            if (string.IsNullOrEmpty(value)) { Invalid(); }
            if (name == "ProductCode" && !Guid.TryParseExact(value, "B", out
_) { Invalid(); }

            if (name == "ProductLanguage" && !ushort.TryParse(value, out _) {
Invalid(); }

            if (name == "ProductVersion")
            {
                string[] parts = value.Split('.');
                if (parts.Length != 3 || !byte.TryParse(parts[0], out _) ||
!byte.TryParse(parts[1], out _) || !ushort.TryParse(parts[2], out _) { Invalid(); }
            }
        }
        // Query actual package columns so an arbitrary compound file or empty MSI
database is insufficient.
        foreach (string query in new[]
        {
            "SELECT `Directory`, `Directory_Parent`, `DefaultDir` FROM
`Directory`,
            "SELECT `Component`, `ComponentId`, `Directory_`, `Attributes`,
`Condition`, `KeyPath` FROM `Component`,
            "SELECT `Feature`, `Feature_Parent`, `Title`, `Description`,
`Display`, `Level`, `Directory_`, `Attributes` FROM `Feature`,
            "SELECT `Feature_`, `Component_` FROM `FeatureComponents`,
            "SELECT `File`, `Component_`, `FileName`, `FileSize`, `Version`,
`Language`, `Attributes`, `Sequence` FROM `File`,
            "SELECT `DiskId`, `LastSequence`, `DiskPrompt`, `Cabinet`,
`VolumeLabel`, `Source` FROM `Media`,
            "SELECT `Action`, `Condition`, `Sequence` FROM
`InstallExecuteSequence`"
        }) { Query(database, query); }
    }
}

```

```

        finally { if (summary != 0) { MsiCloseHandle(summary); } if (database != 0) {
MsiCloseHandle(database); } }
    }

    /// <summary>
    /// Reads a named property from the MSI database.
    /// </summary>
    private static string Property(uint database, string name)
    {
        uint view = 0, record = 0;
        try
        {
            Check(MsiDatabaseOpenView(database, $"SELECT `Value` FROM `Property` WHERE
`Property` = '{name}'", out view));
            Check(MsiViewExecute(view, 0)); Check(MsiViewFetch(view, out record));
            uint length = 1024; StringBuilder value = new(1024);
            Check(MsiRecordGetString(record, 1, value, ref length)); return
value.ToString();
        }
        finally { if (record != 0) { MsiCloseHandle(record); } if (view != 0) {
MsiCloseHandle(view); } }
    }

    /// <summary>
    /// Executes a read-only query to validate the MSI database structure.
    /// </summary>
    private static void Query(uint database, string query)
    {
        uint view = 0;
        try { Check(MsiDatabaseOpenView(database, query, out view));
Check(MsiViewExecute(view, 0)); }
        finally { if (view != 0) { MsiCloseHandle(view); } }
    }

    /// <summary>
    /// Rejects an unsuccessful Windows Installer API result.
    /// </summary>
    private static void Check(uint result) { if (result != 0) { Invalid(); } }

    /// <summary>
    /// Raises an artifact validation error for an invalid installer package.
    /// </summary>
    private static void Invalid() => throw new ArtifactValidationException("The file must be a
Windows Installer package with valid package identity, product properties and standard
installation tables.");

    /// <summary>
    /// Opens a Windows Installer database through the native MSI API.
    /// </summary>
    [DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiOpenDatabaseW")] private
static extern uint MsiOpenDatabase(string path, IntPtr persist, out uint database);

    /// <summary>
    /// Opens the summary information stream of an MSI package.
    /// </summary>
    [DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint =
"MsiGetSummaryInformationW")] private static extern uint MsiGetSummaryInformation(uint database,
string? path, uint update, out uint summary);

    /// <summary>
    /// Reads a property from an MSI summary information stream.
    /// </summary>

```

```

[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint =
"MsiSummaryInfoGetPropertyW")] private static extern uint MsiSummaryInfoGetProperty(uint summary,
uint property, out uint type, out int value, out System.Runtime.InteropServices.ComTypes.FILETIME
time, StringBuilder text, ref uint length);

/// <summary>
/// Creates a native MSI database view for a SQL query.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiDatabaseOpenViewW")]
private static extern uint MsiDatabaseOpenView(uint database, string query, out uint view);

/// <summary>
/// Executes a native MSI database view with the supplied record.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiViewExecute(uint view, uint record);

/// <summary>
/// Fetches the next record from a native MSI database view.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiViewFetch(uint view, out uint
record);

/// <summary>
/// Reads a string field from a native MSI record.
/// </summary>
[DllImport("msi.dll", CharSet = CharSet.Unicode, EntryPoint = "MsiRecordGetStringW")]
private static extern uint MsiRecordGetString(uint record, uint field, StringBuilder value, ref
uint length);

/// <summary>
/// Releases a native Windows Installer handle.
/// </summary>
[DllImport("msi.dll")] private static extern uint MsiCloseHandle(uint handle);
}

```

OperationalRedaction.cs

```

using System.Text.RegularExpressions;
namespace BridgeControl.Services.Vendor;

/// <summary>
/// Detects sensitive operational text before it can be exposed in history.
/// </summary>
public static partial class OperationalRedaction
{
    // Reject the whole free-text value on a secret indicator rather than retaining a
    partially parsed credential.
    /// <summary>
    /// Returns the generated pattern used to detect credential-like log content.
    /// </summary>

    [GeneratedRegex(@"(?:i) (password|passwd|pwd|secret|token|credential|authorization|(?api|access|pr
ivate) [\s_-]*key|connection[\s_-]*string|(?bearer|basic) \s|bcal\.|bcel\.|eyJ[A-Za-z0-9_-]{8,}\.|
-----BEGIN|(?server|data source)\s*=)", RegexOptions.CultureInvariant)]
    private static partial Regex SensitivePattern();

    /// <summary>
    /// Checks whether text contains potentially sensitive credential or connection
    information.
    /// </summary>

```

```

        public static bool IsSensitive(string value) => SensitivePattern().IsMatch(value);
    }

```

RoleGrantPolicy.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Determines which roles an actor may assign within a vendor or customer.
/// </summary>
public class RoleGrantPolicy
{
    private static readonly string[] Known = ["VendorAdministrator", "VendorTechnician",
"CustomerAdministrator", "CustomerUser"];

    /// <summary>
    /// Checks whether the actor is a vendor-wide administrator.
    /// </summary>
    public static bool VendorAdministrator(TenantAccess access) => access.User.CustomerId is
null && access.Roles.Contains("VendorAdministrator");

    /// <summary>
    /// Checks whether the actor is an administrator bound to a customer.
    /// </summary>
    public static bool CustomerAdministrator(TenantAccess access) => access.User.CustomerId is
not null && access.Roles.Contains("CustomerAdministrator");

    /// <summary>
    /// Filters roles to those the current actor may grant.
    /// </summary>
    public IQueryable<Role> Assignable(VendorDbContext db, TenantAccess access)
    {
        if (!VendorAdministrator(access) && !CustomerAdministrator(access)) { throw new
ApiAccessException(403); }
        return db.Roles.Where(role => Known.Contains(role.Name) &&
(VendorAdministrator(access) || role.Scope == "Customer"));
    }

    /// <summary>
    /// Validates requested role grants against the actor's authority and the target customer.
    /// </summary>
    public async Task<List<Role>> ValidateAsync(VendorDbContext db, TenantAccess access, Guid?
customerId, Guid[] roleIds, CancellationToken ct)
    {
        if (!VendorAdministrator(access) && (!CustomerAdministrator(access) || customerId
!= access.User.CustomerId)) { throw new ApiAccessException(404); }
        string scope = customerId is null ? "Vendor" : "Customer";
        List<Role> roles = await Assignable(db, access).Where(x => roleIds.Contains(x.Id)
&& x.Scope == scope).ToListAsync(ct);
        if (roleIds.Distinct().Count() != roleIds.Length || roles.Count != roleIds.Length)
        { throw new ApiConflictException("invalid-role-assignment", "Roles must be known,
distinct and compatible with the user's fixed scope."); }
        return roles;
    }
}

```

SchemaArtifactValidationService.cs

```
using System.Text.Json;
using BridgeControl.Api.Exceptions;

namespace BridgeControl.Services.Vendor;

/// <summary>Selects a registered validator from the schema's $schema declaration.</summary>
public sealed class SchemaArtifactValidationService
{
    private readonly Dictionary<string, ISchemaArtifactValidator> validators =
new(StringComparer.Ordinal);

    /// <summary>
    /// Initializes a new <see cref="SchemaArtifactValidationService"/> instance and its
dependencies.
    /// </summary>
    public SchemaArtifactValidationService(IEnumerable<ISchemaArtifactValidator> validators)
    {
        foreach (ISchemaArtifactValidator validator in validators)
        {
            if (!this.validators.TryAdd(validator.Dialect, validator))
            { throw new InvalidOperationException("Only one schema artifact validator
may be registered per dialect."); }
        }
    }

    /// <summary>
    /// Loads and validates a schema artifact from disk.
    /// </summary>
    public async Task ValidateAsync(string path, CancellationToken ct)
    {
        try
        {
            await using FileStream file = File.OpenRead(path);
            using JsonDocument document = await JsonDocument.ParseAsync(file, new() {
MaxDepth = 64 }, ct);
            Validate(document.RootElement, ct);
        }
        catch (JsonException) { throw new ArtifactValidationException("Schema must be
complete UTF-8 JSON with a maximum depth of 64."); }
    }

    /// <summary>
    /// Validates schema JSON using the configured dialect validator.
    /// </summary>
    public void Validate(JsonElement schema, CancellationToken ct)
    {
        ct.ThrowIfCancellationRequested();
        if (schema.ValueKind != JsonValueKind.Object || !schema.TryGetProperty("$schema",
out JsonElement dialect)
            || dialect.ValueKind != JsonValueKind.String ||
string.IsNullOrEmpty(dialect.GetString()))
        { throw new ArtifactValidationException("Declare the schema dialect using
$schema."); }
        if (!validators.TryGetValue(dialect.GetString(), out ISchemaArtifactValidator?
validator))
        { throw new ArtifactValidationException("No validator is registered for the
declared schema dialect."); }
    }
}
```



```

        validator.Validate(schema, ct);
    }
}

```

StateReconciliationProcessor.cs

```

using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Repositories.Vendor;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>Creates durable work from accepted actual evidence. Command success never
substitutes for actual state.</summary>
public sealed class StateReconciliationProcessor(PlatformDbContext platform, ITenantResolver
resolver, InstallationStatePolicy policy, VendorSecurityAudit audit, TimeProvider clock) :
IVendorProcessor
{
    /// <summary>
    /// Processes pending reconciliation work for one vendor with bounded retries.
    /// </summary>
    public async Task ProcessVendorAsync(Guid vendorId, CancellationToken ct = default)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await resolver.ResolveForAgentAsync(vendorId,
ct); if (tenant is null) { return; }
        VendorDbContext db = tenant.Database; DateTimeOffset now = clock.GetUtcNow();
        Guid[] ids = await db.StateReconciliationWork.Where(x => (x.Status == "Pending" ||
x.Status == "Blocked") && x.NextAttemptAt <= now)
        .OrderBy(x => x.NextAttemptAt).ThenBy(x => x.InstallationId).Select(x =>
x.InstallationId).Take(20).ToArrayAsync(ct);
        foreach (Guid id in ids)
        {
            db.ChangeTracker.Clear();
            try { await ReconcileAsync(tenant, id, ct); }
            catch (OperationCanceledException) when (ct.IsCancellationRequested) {
throw; }

            catch (Exception error)
            {
                db.ChangeTracker.Clear(); StateReconciliationWork work = await
db.StateReconciliationWork.SingleAsync(x => x.InstallationId == id, ct);
                work.Attempts++; work.Status = work.Attempts >= 3 ? "Failed" :
"Pending";
                work.Reason = "ReconciliationFailed:" + error.GetType().Name;
                work.UpdatedAt = clock.GetUtcNow(); work.NextAttemptAt = work.UpdatedAt.AddSeconds(60 *
work.Attempts);

                await db.SaveChangesAsync(ct);
            }
        }

        /// <summary>
        /// Compares desired and reported state and updates deployment work for one installation.
        /// </summary>
    }
}

```



```

        RequestedAt = clock.GetUtcNow()
    };
    db.Deployments.Add(deployment); audit.Record(db,
Guid.Empty, "DeploymentPlanned", installationId, entityType: "Installation", actorType: "System",
resourceId: deployment.Id);
    }
    else if (active is not null && active.OperationType !=
classification)
    {
        // An operation's execution semantics are fixed once
an attempt has started.
        bool started = await
db.DeploymentAttempts.AnyAsync(x => x.DeploymentId == active.Id, ct);
        if (!started) { active.OperationType =
classification; }
    }
    work.Status = "Succeeded";
}
}
}
catch (Exception error) when (error is ApiConflictException or
ConfigurationValidationException or ApiAccessException)
{ work.Status = "Blocked"; work.Reason = error is ApiConflictException
conflict ? conflict.Code : "TargetUnavailable"; }
}
work.CompletedGeneration = work.RequestedGeneration; work.UpdatedAt =
clock.GetUtcNow(); work.NextAttemptAt = work.UpdatedAt.AddSeconds(60);
await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
}

/// <summary>
/// Classifies reported installation evidence against the desired target and current time.
/// </summary>
public static string Classify(InstallationStateReport? report, StateTarget target,
DateTimeOffset now)
{
    if (report is null || report.ReportedAt < now.AddMinutes(-15) || report.Status ==
"Unknown") { return "Unknown"; }
    if (report.Status == "NotInstalled") { return "Install"; }
    if (report.InstalledProductVersionId is null) { return "Unknown"; }
    if (report.InstalledProductVersionId != target.Version.Id) { return "Update"; }
    if (report.Status is "Running" or "Ready" &&
report.EffectiveConfigurationRevisionId == target.Configuration.Id
&& report.ConfigurationHash == target.Configuration.FileArtifact.Sha256) {
return "Matching"; }
    return "Configure";
}
}
}

```

StateReconciliationWorker.cs

```

using BridgeControl.Options;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>Runs recurring state reconciliation work until the host stops.</summary>
public sealed class StateReconciliationWorker(IServiceScopeFactory scopes,
IOptions<StateReconciliationOptions> configured,
ILogger<StateReconciliationWorker> logger) : VendorWorker<StateReconciliationProcessor>(

```

```
scopes, () => configured.Value.WorkerEnabled, () => configured.Value.PollSeconds,
logger, "State reconciliation");
```

TenantAccess.cs

```
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>Current database grants and reusable query boundaries, applied before paging or
projection.</summary>
public sealed class TenantAccess(UserAccount user, IReadOnlyList<string> roles,
IReadOnlyList<string> permissions)
{
    public UserAccount User { get; } = user;
    public IReadOnlyList<string> Roles { get; } = roles;
    public IReadOnlyList<string> Permissions { get; } = permissions;

    /// <summary>
    /// Checks whether the current user's permissions include the requested code.
    /// </summary>
    public bool HasPermission(string code) => Permissions.Contains(code);
    private bool VendorAdministrator => User.CustomerId is null &&
Roles.Contains("VendorAdministrator");
    private bool Technician => User.CustomerId is null && Roles.Contains("VendorTechnician");
    private bool CustomerAdministrator => User.CustomerId is not null &&
Roles.Contains("CustomerAdministrator");

    /// <summary>
    /// Filters customers to those visible under the user's permissions and assignments.
    /// </summary>
    public IQueryable<Customer> Customers(VendorDbContext db, string permission =
"customers.read") => db.Customers.Where(customer => HasPermission(permission) && (
VendorAdministrator || (User.CustomerId != null && customer.Id == User.CustomerId)
|| (Technician && db.TechnicianCustomerAccessGrants.Any(grant => grant.UserId ==
User.Id && grant.CustomerId == customer.Id))));

    /// <summary>
    /// Filters installations to those visible under the user's permissions and assignments.
    /// </summary>
    public IQueryable<Installation> Installations(VendorDbContext db, string permission =
"installations.read") => db.Installations.Where(installation => HasPermission(permission) && (
VendorAdministrator || (Technician && db.TechnicianCustomerAccessGrants.Any(grant
=> grant.UserId == User.Id && grant.CustomerId == installation.CustomerId)
|| (installation.CustomerId == User.CustomerId && (CustomerAdministrator
|| db.CustomerUserInstallationAccessGrants.Any(grant => grant.UserId ==
User.Id && grant.CustomerId == User.CustomerId && grant.InstallationId == installation.Id))));

    /// Operational actions must also honor customer lifecycle; ordinary reads can retain
disabled history.
    /// <summary>
    /// Filters installations to the operational scope allowed for the current user.
    /// </summary>
    public IQueryable<Installation> OperationalInstallations(VendorDbContext db, string
permission = "installations.manage")
=> Installations(db, permission).Where(installation =>
installation.Customer.Status == "Active" && installation.Status == "Active");
```

```

    /// <summary>
    /// Filters user accounts to those visible under the current user's authority.
    /// </summary>
    public IQueryable<UserAccount> Users(VendorDbContext db, string permission = "users.read")
=> db.UserAccounts.Where(account => HasPermission(permission) && (
    VendorAdministrator || (CustomerAdministrator && account.CustomerId ==
User.CustomerId)));

    /// <summary>
    /// Loads an active user's roles, permissions, and resource assignments.
    /// </summary>
    public static async Task<TenantAccess?> LoadAsync(VendorDbContext db, Guid userId,
CancellationToken ct)
    {
        UserAccount? user = await db.UserAccounts.Include(x =>
x.Customer).AsNoTracking().SingleOrDefaultAsync(x => x.Id == userId, ct);
        if (user?.Status != "Active" || (user.CustomerId is not null &&
user.Customer?.Status != "Active")) { return null; }
        string scope = user.CustomerId is null ? "Vendor" : "Customer";
        string[] roles = await db.UserRoles.Where(x => x.UserId == userId && x.Role.Scope
== scope).Select(x => x.Role.Name).Distinct().ToArrayAsync(ct);
        string[] permissions = await db.RolePermissions.Where(x => x.Role.Scope == scope
&& x.Role.UserRoles.Any(grant => grant.UserId == userId))
        .Select(x => x.Permission.Code).Distinct().ToArrayAsync(ct);
        return new TenantAccess(user, roles, permissions);
    }
}

```

TenantAuthenticationService.cs

```

using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using BridgeControl.Data;
using BridgeControl.DTOS.Vendor.TenantAuthenticationDtos;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;
using Microsoft.IdentityModel.Tokens;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents tenant authentication result data.
/// </summary>
public record TenantAuthenticationResult(AuthSessionResponse Response, string RefreshCredential,
DateTimeOffset RefreshExpiresAt);

/// <summary>
/// Coordinates tenant authentication operations.
/// </summary>
public class TenantAuthenticationService(PlatformDbContext platform, ITenantResolver resolver,
PasswordService<UserAccount> passwords, AuthenticationSessionService sessions,
VendorSecurityAudit audit,
IOptions<TenantAuthenticationOptions> configuration, TimeProvider clock)
{

```

```

    /// <summary>
    /// Validates tenant credentials and issues an authentication session for an active user.
    /// </summary>
    public async Task<TenantAuthenticationResult?> LoginAsync(LoginRequest request,
CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await
resolver.ResolveForLoginAsync(request.VendorName, ct);
        string email = AccountValidation.NormalizeEmail(request.Email);
        UserAccount? user = tenant is null ? null : await
tenant.Database.UserAccounts.SingleOrDefaultAsync(x => x.Email == email, ct);
        PasswordVerificationResult verification = passwords.Verify(user,
user?.PasswordHash, request.Password);
        TenantAccess? access = tenant is null || user is null ? null : await
TenantAccess.LoadAsync(tenant.Database, user.Id, ct);
        if (tenant is null || access is null || verification ==
PasswordVerificationResult.Failed)
        {
            if (tenant is not null)
            {
                audit.Record(tenant.Database, user?.Id ?? Guid.Empty,
"LoginFailed", user?.Id ?? Guid.Empty, "Denied");
                await tenant.Database.SaveChangesAsync(ct);
            }
            return null;
        }
        if (verification == PasswordVerificationResult.SuccessRehashNeeded) {
user!.PasswordHash = passwords.Hash(user, request.Password); }
        user!.LastLoginAt = clock.GetUtcNow();
        TenantAuthenticationOptions options = configuration.Value;
        IssuedSession issued = sessions.Create(user.Id,
TimeSpan.FromDays(options.SessionDays), TimeSpan.FromDays(options.RefreshTokenDays));
        tenant.Database.AuthenticationSessions.Add(issued.Session);
        audit.Record(tenant.Database, user.Id, "SessionCreated", issued.Session.Id,
entityType: "AuthenticationSession");
        await tenant.Database.SaveChangesAsync(ct);
        return Create(tenant.VendorId, access, issued);
    }

    /// <summary>
    /// Rotates a valid tenant refresh credential and returns a renewed session.
    /// </summary>
    public async Task<TenantAuthenticationResult?> RefreshAsync(string? credential,
CancellationToken ct)
    {
        string[] parts = credential?.Split('.') ?? [];
        if (parts.Length != 2 || !Guid.TryParseExact(parts[0], "N", out Guid vendorId)) {
return null; }
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await ResolveAsync(vendorId, ct);
        if (tenant is null) { return null; }
        VendorDbContext db = tenant.Database;
        // A vendor disable cutoff is authoritative even if physical session cleanup was
interrupted.
        DateTimeOffset? cutoff = await platform.SoftwareVendors.Where(x => x.Id ==
vendorId).Select(x => x.SessionsRevokedBefore).SingleAsync(ct);
        if (cutoff is not null)
        {

```

```

        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        int count = await db.AuthenticationSessions.Where(x => x.RevokedAt == null
&& x.CreatedAt <= cutoff).ExecuteUpdateAsync(s => s.SetProperty(x => x.RevokedAt, cutoff), ct);
        if (count > 0) { audit.Record(db, Guid.Empty, "VendorSessionsRevoked",
vendorId, entityType: "SoftwareVendor", actorType: "System"); }
        await db.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
    }
    IssuedSession? issued = await sessions.RotateAsync(db, parts[1],
(TimeSpan.FromDays(configuration.Value.RefreshTokenDays),
        async (id, token) => await TenantAccess.LoadAsync(db, id, token) is not
null,
        (session, action) => audit.Record(db, session.AccountId, action,
session.Id, entityType: "AuthenticationSession"), ct);
        if (issued is null) { return null; }
        TenantAccess access = (await TenantAccess.LoadAsync(db, issued.Session.AccountId,
ct))!;
        return Create(vendorId, access, issued);
    }

    /// <summary>
    /// Resolves the authenticated vendor context for session operations.
    /// </summary>
    public Task<TenantContext?> ResolveAsync(Guid vendorId, CancellationToken ct) =>
resolver.ResolveAuthenticatedAsync(
    new ClaimsPrincipal(new ClaimsIdentity([new Claim("identity_type", "vendor_user"),
new Claim("vendor_id", vendorId.ToString())], "internal")), ct);

    /// <summary>
    /// Rechecks live account, session, and access state for an authenticated tenant user.
    /// </summary>
    public async Task<TenantAccess?> ValidateAsync(TenantContext tenant, Guid userId, Guid
sessionId, CancellationToken ct)
    {
        DateTimeOffset now = clock.GetUtcNow();
        DateTimeOffset? cutoff = await platform.SoftwareVendors.AsNoTracking().Where(x =>
x.Id == tenant.VendorId).Select(x => x.SessionsRevokedBefore).SingleAsync(ct);
        bool valid = await
tenant.Database.AuthenticationSessions.AsNoTracking().AnyAsync(x => x.Id == sessionId &&
x.AccountId == userId
        && x.RevokedAt == null && x.ExpiresAt > now && (cutoff == null ||
x.CreatedAt > cutoff), ct);
        return valid ? await TenantAccess.LoadAsync(tenant.Database, userId, ct) : null;
    }

    /// <summary>
    /// Revokes the specified tenant user's authentication session.
    /// </summary>
    public async Task<bool> LogoutAsync(Guid vendorId, Guid userId, Guid sessionId,
CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await ResolveAsync(vendorId, ct);
        if (tenant is null || await ValidateAsync(tenant, userId, sessionId, ct) is null)
{ return false; }
        await sessions.RevokeAsync(tenant.Database, sessionId, userId,
            (session, action) => audit.Record(tenant.Database, userId, action,
session.Id, entityType: "AuthenticationSession"), "SessionLoggedOut", ct);
        return true;
    }
}

```

```

    /// <summary>
    /// Builds the tenant access token and session response from an issued session.
    /// </summary>
    private TenantAuthenticationResult Create(Guid vendorId, TenantAccess access,
IssuedSession issued)
    {
        TenantAuthenticationOptions options = configuration.Value;
        DateTimeOffset now = clock.GetUtcNow();
        DateTimeOffset expiry = now.AddMinutes(options.AccessTokenMinutes);
        if (expiry > issued.Session.ExpiresAt) { expiry = issued.Session.ExpiresAt; }
        expiry = DateTimeOffset.FromUnixTimeSeconds(expiry.ToUnixTimeSeconds());
        Claim[] claims = [new("sub", access.User.Id.ToString()), new("sid",
issued.Session.Id.ToString()),
            new("vendor_id", vendorId.ToString()), new("identity_type", "vendor_user"),
new("jti", Guid.NewGuid().ToString())];
        JwtSecurityToken token = new JwtSecurityToken(options.Issuer, options.Audience,
claims, now.UtcDateTime, expiry.UtcDateTime,
            new SigningCredentials(new
SymmetricSecurityKey(Convert.FromBase64String(options.SigningKey)),
SecurityAlgorithms.HmacSha256));
        token.Header["typ"] = "at+jwt";
        return new(new AuthSessionResponse
        {
            UserId = access.User.Id,
            VendorId = vendorId,
            Email = access.User.Email,
            CustomerId = access.User.CustomerId,
            Roles = access.Roles,
            Permissions = access.Permissions,
            AccessToken = new JwtSecurityTokenHandler().WriteToken(token),
            AccessTokenExpiresAt = expiry
        },
            $"{vendorId:N}.{issued.RefreshCredential}", issued.RefreshExpiresAt);
    }
}

```

TenantOperation.cs

```

using System.Security.Claims;
using BridgeControl.Api.Exceptions;
using BridgeControl.Data;
using BridgeControl.Security;
using BridgeControl.Services.Platform;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Represents tenant actor data.
/// </summary>
public record TenantActor(Guid VendorId, Guid UserId, Guid SessionId)
{
    /// <summary>
    /// Creates an actor identity from the authenticated principal's claims.
    /// </summary>
    public static TenantActor From(ClaimsPrincipal principal) =>
new(TenantAuthentication.VendorId(principal), TenantAuthentication.UserId(principal),
TenantAuthentication.SessionId(principal));
}

```



```

/// <summary>Owns the trusted database and optional write lock for one authorized
operation.</summary>
public sealed class TenantOperation(TenantContext tenant, TenantAccess access,
PlatformVendorLock? writeLock) : IAsyncDisposable
{
    public TenantContext Tenant { get; } = tenant;
    public VendorDbContext Database => Tenant.Database;
    public TenantAccess Access { get; } = access;
    /// <summary>
    /// Asynchronously releases the resources owned by this instance.
    /// </summary>
    public async ValueTask DisposeAsync()
    {
        try { await Tenant.DisposeAsync(); }
        finally { if (writeLock is not null) { await writeLock.DisposeAsync(); } }
    }
}

/// <summary>
/// Creates authorized tenant operations with the required database locks.
/// </summary>
public class TenantOperationFactory(PlatformDbContext platform, TenantAuthenticationService
authentication)
{
    /// <summary>
    /// Opens an authorized tenant operation and acquires the required write lock.
    /// </summary>
    public async Task<TenantOperation> OpenAsync(TenantActor actor, string permission, bool
write, CancellationToken ct)
    {
        PlatformVendorLock? writeLock = write ? await
PlatformVendorLock.AcquireAsync(platform, ct) : null;
        TenantContext? tenant = null;
        try
        {
            tenant = await authentication.ResolveAsync(actor.VendorId, ct);
            TenantAccess? access = tenant is null ? null : await
authentication.ValidateAsync(tenant, actor.UserId, actor.SessionId, ct);
            if (tenant is null || access is null) { throw new ApiAccessException(401);
}

            if (!access.HasPermission(permission)) { throw new ApiAccessException(403);
}

            return new TenantOperation(tenant, access, writeLock);
        }
        catch
        {
            if (tenant is not null) { await tenant.DisposeAsync(); }
            if (writeLock is not null) { await writeLock.DisposeAsync(); }
            throw;
        }
    }
}

```

TenantPasswordService.cs

```

using System.Security.Cryptography;
using BridgeControl.Data;
using BridgeControl.DTOS.Vendor.TenantAuthenticationDtos;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;

```

```

using BridgeControl.Services.Platform;
using BridgeControl.Services.Shared;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Defines the contract for password reset delivery operations.
/// </summary>
public interface IPasswordResetDelivery
{
    /// <summary>
    /// Delivers a password-reset credential with its expiry time.
    /// </summary>
    Task DeliverAsync(Guid id, string email, string token, DateTimeOffset expiresAt,
CancellationToken ct);
}

/// <summary>
/// Delivers password-reset invitations through the protected local outbox.
/// </summary>
public class LocalPasswordResetDelivery(IOptions<VendorProvisioningOptions> options) :
IPasswordResetDelivery
{
    /// <summary>
    /// Writes a password-reset invitation through the local delivery service.
    /// </summary>
    public Task DeliverAsync(Guid id, string email, string token, DateTimeOffset expiresAt,
CancellationToken ct)
=> new LocalInvitationDelivery(options).DeliverResetAsync(id, email, token,
expiresAt, ct);
}

/// <summary>
/// Issues recoverable password-reset invitations and revokes credentials when passwords change.
/// </summary>
public class TenantPasswordService(PlatformDbContext platform, ITenantResolver resolver,
TenantAuthenticationService authentication,
    PasswordService<UserAccount> passwords, RefreshCredentialService credentials,
IDataProtectionProvider protection,
    IPasswordResetDelivery delivery, VendorSecurityAudit audit, TimeProvider clock)
{
    /// <summary>
    /// Issues a password-reset invitation without revealing whether an account exists.
    /// </summary>
    public async Task ForgotAsync(ForgotPasswordRequest request, CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await
resolver.ResolveForLoginAsync(request.VendorName, ct);
        if (tenant is null)
        {
            return;
        }

        VendorDbContext database = tenant.Database;
        string email = AccountValidation.NormalizeEmail(request.Email);

```

```

        UserAccount? user = await database.UserAccounts.SingleOrDefaultAsync(account =>
account.Email == email, ct);
        if (user is null || await TenantAccess.LoadAsync(database, user.Id, ct) is null)
        {
            audit.Record(database, user?.Id ?? Guid.Empty, "PasswordResetRequested",
user?.Id ?? Guid.Empty, "Denied");
            await database.SaveChangesAsync(ct);
            return;
        }

        IDataProtector protector =
protection.CreateProtector("BridgeControl.PasswordResetDelivery.v1",
tenant.VendorId.ToString("N"));
        PasswordReset reset = await GetOrCreateResetAsync(database, tenant.VendorId, user,
protector, ct);
        await DeliverResetAsync(database, user, email, reset, protector, ct);
    }

    /// <summary>
    /// Consumes a valid reset credential and replaces the account password.
    /// </summary>
    public async Task<bool> ResetAsync(ResetPasswordRequest request, CancellationToken ct)
    {
        string[] parts = request.Token.Split('.');
        if (parts.Length != 4 || parts[0] != "reset" || !Guid.TryParseExact(parts[1], "N",
out Guid vendorId)
            || !Guid.TryParseExact(parts[2], "N", out Guid resetId)
            || !credentials.TryHash(parts[3], out byte[] hash))
        {
            return false;
        }
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await authentication.ResolveAsync(vendorId,
ct);
        if (tenant is null)
        {
            return false;
        }

        VendorDbContext database = tenant.Database;
        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(ct);
        PasswordReset? reset = await database.PasswordResets
            .Include(reset => reset.User)
            .SingleOrDefaultAsync(reset => reset.Id == resetId, ct);
        if (reset is null || !IsUsableReset(reset, hash)
            || await TenantAccess.LoadAsync(database, reset.UserId, ct) is null
            || AccountValidation.ValidateNewAccount(reset.User.Email, request.Password)
is not null)
        {
            audit.Record(database, reset?.UserId ?? Guid.Empty, "PasswordResetFailed",
reset?.UserId ?? Guid.Empty, "Denied");
            await database.SaveChangesAsync(ct);
            await transaction.CommitAsync(ct);
            return false;
        }

        reset.UsedAt = clock.GetUtcNow();
        reset.ProtectedToken = null;
        reset.User.PasswordHash = passwords.Hash(reset.User, request.Password);
        await database.SaveChangesAsync(ct);
    }

```

```

        await RevokeCredentialsAsync(database, reset.UserId, ct);
        audit.Record(database, reset.UserId, "PasswordReset", reset.UserId);
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
        return true;
    }

    /// <summary>
    /// Changes an authenticated user's password after verifying the current password.
    /// </summary>
    public async Task<bool> ChangeAsync(Guid vendorId, Guid userId, Guid sessionId,
    ChangePasswordRequest request, CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
    PlatformVendorLock.AcquireAsync(platform, ct);
        await using TenantContext? tenant = await authentication.ResolveAsync(vendorId,
    ct);
        if (tenant is null || await authentication.ValidateAsync(tenant, userId,
    sessionId, ct) is null)
        {
            return false;
        }

        VendorDbContext database = tenant.Database;
        await using IDbContextTransaction transaction = await
    database.Database.BeginTransactionAsync(ct);
        UserAccount user = await database.UserAccounts.SingleAsync(account => account.Id
    == userId, ct);
        if (passwords.Verify(user, user.PasswordHash, request.CurrentPassword) ==
    PasswordVerificationResult.Failed
            || AccountValidation.ValidateNewAccount(user.Email, request.NewPassword) is
    not null)
        {
            audit.Record(database, userId, "PasswordChangeFailed", userId, "Denied");
            await database.SaveChangesAsync(ct);
            await transaction.CommitAsync(ct);
            return false;
        }

        user.PasswordHash = passwords.Hash(user, request.NewPassword);
        await RevokeCredentialsAsync(database, userId, ct);
        audit.Record(database, userId, "PasswordChanged", userId);
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
        return true;
    }

    /// <summary>
    /// Revokes a user's sessions and outstanding account credentials.
    /// </summary>
    public async Task RevokeCredentialsAsync(VendorDbContext database, Guid userId,
    CancellationToken ct, Guid? actorId = null)
    {
        DateTimeOffset now = clock.GetUtcNow();
        await database.AuthenticationSessions
            .Where(session => session.AccountId == userId && session.RevokedAt == null)
            .ExecuteUpdateAsync(setters => setters.SetProperty(session =>
    session.RevokedAt, now), ct);
        await database.PasswordResets
            .Where(reset => reset.UserId == userId && reset.UsedAt == null &&
    reset.RevokedAt == null)
            .ExecuteUpdateAsync(setters => setters

```

```

        .SetProperty(reset => reset.RevokedAt, now)
        .SetProperty(reset => reset.ProtectedToken, (string?)null), ct);
    await database.AccountActivations
        .Where(activation => activation.UserId == userId && activation.UsedAt ==
null && activation.RevokedAt == null)
        .ExecuteUpdateAsync(setters => setters
            .SetProperty(activation => activation.RevokedAt, now)
            .SetProperty(activation => activation.ProtectedToken,
(string?)null), ct);
    audit.Record(database, actorId ?? userId, "AccountCredentialsRevoked", userId);
}

/// <summary>Commits retryable reset state before attempting delivery outside the
transaction.</summary>
private async Task<PasswordReset> GetOrCreateResetAsync(
    VendorDbContext database, Guid vendorId, UserAccount user, IDataProtector
protector, CancellationToken ct)
{
    PasswordReset reset;
    await using (IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(ct))
    {
        PasswordReset? pending = await database.PasswordResets
            .SingleOrDefaultAsync(reset => reset.UserId == user.Id
                && reset.UsedAt == null && reset.RevokedAt == null, ct);
        if (pending is not null && pending.ExpiresAt <= clock.GetUtcNow())
        {
            pending.RevokedAt = clock.GetUtcNow();
            pending.ProtectedToken = null;
            await database.SaveChangesAsync(ct);
            pending = null;
        }
        if (pending is null)
        {
            string secret = credentials.Create();
            credentials.TryHash(secret, out byte[] hash);
            Guid id = Guid.NewGuid();
            reset = new PasswordReset
            {
                Id = id,
                UserId = user.Id,
                TokenHash = hash,
                CreatedAt = clock.GetUtcNow(),
                ExpiresAt = clock.GetUtcNow().AddMinutes(30),
                ProtectedToken =
protector.Protect($"reset.{vendorId:N}.{id:N}.{secret}")
            };
            database.PasswordResets.Add(reset);
        }
        else
        {
            reset = pending;
        }
        audit.Record(database, user.Id, "PasswordResetRequested", user.Id);
        await database.SaveChangesAsync(ct);
        await transaction.CommitAsync(ct);
    }
    return reset;
}

private async Task DeliverResetAsync(VendorDbContext database, UserAccount user, string
email,

```

```

        PasswordReset reset, IDataProtector protector, CancellationToken ct)
    {
        if (reset.DeliveredAt is not null)
        {
            return;
        }
        try
        {
            await delivery.DeliverAsync(reset.Id, email,
protector.Unprotect(reset.ProtectedToken!), reset.ExpiresAt, ct);
            reset.DeliveredAt = clock.UtcNow();
            reset.ProtectedToken = null;
            audit.Record(database, user.Id, "PasswordResetDelivered", user.Id);
            await database.SaveChangesAsync(ct);
        }
        catch (Exception) when (!ct.IsCancellationRequested)
        {
            // Discard delivery-only changes; the committed protected token remains
available for a retry.
            database.ChangeTracker.Clear();
            audit.Record(database, user.Id, "PasswordResetDeliveryFailed", user.Id,
"Failed");
            await database.SaveChangesAsync(ct);
        }
    }

    private bool IsUsableReset>PasswordReset reset, byte[] tokenHash)
        => reset.UsedAt is null
            && reset.RevokedAt is null
            && reset.DeliveredAt is not null
            && reset.ExpiresAt > clock.UtcNow()
            && CryptographicOperations.FixedTimeEquals(reset.TokenHash, tokenHash);
}

```

UserLifecycleService.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using BridgeControl.Services.Platform;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Coordinates user lifecycle operations.
/// </summary>
public class UserLifecycleService(PlatformDbContext platform, TenantAuthenticationService
authentication,
    TenantPasswordService credentials, VendorSecurityAudit audit)
{
    /// <summary>
    /// Changes a user's active status and revokes credentials when access is disabled.
    /// </summary>
    public async Task<int> SetActiveAsync(Guid vendorId, Guid actorId, Guid sessionId, Guid
targetId, bool active, CancellationToken ct)
    {
        await using PlatformVendorLock operation = await
PlatformVendorLock.AcquireAsync(platform, ct);
    }
}

```

```

        await using TenantContext? tenant = await authentication.ResolveAsync(vendorId,
ct);
        TenantAccess? access = tenant is null ? null : await
authentication.ValidateAsync(tenant, actorId, sessionId, ct);
        if (tenant is null || access is null) { return 401; }
        if (!access.HasPermission("users.manage")) { return 403; }
        VendorDbContext db = tenant.Database;
        UserAccount? target = await access.Users(db,
"users.manage").SingleOrDefaultAsync(x => x.Id == targetId, ct);
        if (target is null) { return 404; }
        await using IDbContextTransaction transaction = await
db.Database.BeginTransactionAsync(ct);
        string desired = active ? (string.IsNullOrEmpty(target.PasswordHash) ? "Invited" :
"Active") : "Disabled";
        bool changed = target.Status != desired;
        if (changed) { target.Status = desired; }
        if (!active) { await credentials.RevokeCredentialsAsync(db, targetId, ct,
actorId); }
        audit.Record(db, actorId, active ? "UserReactivated" : "UserDeactivated",
targetId, changed ? "Succeeded" : "Unchanged");
        await db.SaveChangesAsync(ct); await transaction.CommitAsync(ct);
        return 204;
    }
}

```

VendorArtifactArchive.cs

```

using System.Buffers;
using System.Security.Cryptography;
using BridgeControl.Api.Exceptions;
using BridgeControl.Models.Vendor;
using BridgeControl.Options;
using BridgeControl.Services.Shared;
using Microsoft.Extensions.Options;

namespace BridgeControl.Services.Vendor;

/// <summary>Private, immutable blobs addressed only through trusted tenant metadata.</summary>
public class VendorArtifactArchive(IOptions<ArtifactArchiveOptions> options,
SchemaArtifactValidationService schemas)
{
    /// <summary>
    /// Returns the maximum accepted byte length for an artifact kind.
    /// </summary>
    public long Limit(string kind) => kind == "Installer" ? options.Value.InstallerMaxBytes :
kind == "Configuration" ? ConfigurationValues.MaximumBytes : options.Value.SchemaMaxBytes;
    public TimeSpan UploadTimeout => TimeSpan.FromSeconds(options.Value.UploadTimeoutSeconds);

    /// <summary>
    /// Stages and validates an uploaded artifact before it is published.
    /// </summary>
    public Task<StagedArtifact> StageAsync(string archivePath, string kind, string fileName,
Stream source, CancellationToken ct)
    => StageCoreAsync(archivePath, kind, fileName, source, ct);

    /// <summary>
    /// Stages canonical configuration bytes as a configuration artifact.
    /// </summary>
    internal async Task<StagedArtifact> StageConfigurationAsync(string archivePath, byte[]
canonical, CancellationToken ct)

```

```

        {
            using MemoryStream source = new(canonical, writable: false);
            return await StageCoreAsync(archivePath, "Configuration", "configuration.json",
source, ct);
        }

        /// <summary>
        /// Copies an artifact into bounded temporary storage and records its digest and metadata.
        /// </summary>
        private async Task<StagedArtifact> StageCoreAsync(string archivePath, string kind, string
fileName, Stream source, CancellationToken ct)
        {
            ValidateName(fileName, kind);
            string root = PrivateDirectory.Ensure(archivePath);
            string stageDirectory = PrivateDirectory.Ensure(Path.Combine(root, "staging"));
            string blobDirectory = PrivateDirectory.Ensure(Path.Combine(root, "blobs"));
            CleanupAbandonedStages(stageDirectory);
            string identifier = Guid.NewGuid().ToString("N");
            string path = Path.Combine(stageDirectory, identifier + ".upload");
            StagedArtifact staged = new(path, Path.Combine(blobDirectory, identifier +
".blob"), identifier, fileName,
                kind == "Installer" ? "application/x-msi" : kind == "Schema" ?
"application/schema+json" : "application/json");
            try
            {
                using IncrementalHash hash =
IncrementalHash.CreateHash(HashAlgorithmName.SHA256);
                byte[] buffer = ArrayPool<byte>.Shared.Rent(81920);
                try
                {
                    await using FileStream output = new(path, FileMode.CreateNew,
FileAccess.Write, FileShare.None, 81920, FileOptions.Asynchronous);
                    int count;
                    while ((count = await source.ReadAsync(buffer.AsMemory(0, 81920),
ct)) != 0)
                    {
                        staged.SizeBytes += count;
                        if (staged.SizeBytes > Limit(kind)) { throw new
ArtifactValidationException($"The {kind.ToLowerInvariant()} exceeds the configured {Limit(kind)}
byte limit."); }
                        hash.AppendData(buffer, 0, count);
                        await output.WriteAsync(buffer.AsMemory(0, count), ct);
                    }
                    if (staged.SizeBytes == 0) { throw new
ArtifactValidationException("The artifact cannot be empty."); }
                    await output.FlushAsync(ct); output.Flush(flushToDisk: true);
                    staged.Sha256 = Convert.ToHexStringLower(hash.GetHashAndReset());
                }
                finally { ArrayPool<byte>.Shared.Return(buffer, clearArray: true); }
                ct.ThrowIfCancellationRequested();
                if (kind == "Installer")
                {
                    MsiArtifactValidator.Validate(path);
                    staged.InstallerIdentityJson =
System.Text.Json.JsonSerializer.Serialize(MsiArtifactValidator.ReadIdentity(path));
                }
                else if (kind == "Schema") { await schemas.ValidateAsync(path, ct); }
                ct.ThrowIfCancellationRequested();
                return staged;
            }
            catch { await staged.DisposeAsync(); throw; }
        }
    }

```



```

    /// <summary>
    /// Opens an archived artifact after checking its stored identity and integrity.
    /// </summary>
    public async Task<Stream> OpenAsync(string archivePath, FileArtifact artifact,
CancellationTokens ct)
    {
        if (!Guid.TryParseExact(artifact.StorageIdentifier, "N", out Guid id) ||
id.ToString("N") != artifact.StorageIdentifier)
        { throw new InvalidOperationException("Invalid archive metadata."); }
        string root = PrivateDirectory.Ensure(archivePath);
        string directory = PrivateDirectory.Ensure(Path.Combine(root, "blobs"));
        string path = Path.Combine(directory, id.ToString("N") + ".blob");
        if ((File.GetAttributes(path) & FileAttributes.ReparsePoint) != 0) { throw new
InvalidOperationException("Unsafe archive entry."); }
        FileStream stream = new(path, FileMode.Open, FileAccess.Read, FileShare.Read,
81920, FileOptions.Asynchronous);
        try
        {
            if (stream.Length != artifact.SizeBytes || Convert.ToHexStringLower(await
SHA256.HashDataAsync(stream, ct)) != artifact.Sha256)
            { throw new InvalidOperationException("Archive integrity check failed."); }
            stream.Position = 0; return stream;
        }
        catch { await stream.DisposeAsync(); throw; }
    }

    /// <summary>
    /// Rejects unsafe filenames and extensions that do not match the artifact kind.
    /// </summary>
    private static void ValidateName(string fileName, string kind)
    {
        string extension = kind == "Installer" ? ".msi" : ".json";
        if (string.IsNullOrEmpty(fileName) || fileName.Length > 255 || fileName !=
fileName.Trim()
            || fileName.Any(c => char.IsControl(c) || "<>:\/\\|?*".Contains(c)) ||
fileName.EndsWith('.')
            || !fileName.EndsWith(extension, StringComparison.OrdinalIgnoreCase))
        { throw new ArtifactValidationException($"Supply a plain {extension} filename
without paths or unsafe characters."); }
    }

    /// <summary>
    /// Removes expired temporary artifact files left by interrupted uploads.
    /// </summary>
    private static void CleanupAbandonedStages(string directory)
    {
        // Uploads time out within five minutes. Only generated, unlocked stages older
        than a day are eligible.
        foreach (string path in Directory.EnumerateFiles(directory, "*.upload").Take(256))
        {
            if (!Guid.TryParseExact(Path.GetFileNameWithoutExtension(path), "N", out
_) ) { continue; }

            FileInfo file = new(path);
            if ((file.Attributes & FileAttributes.ReparsePoint) != 0 ||
file.LastWriteTimeUtc >= DateTime.UtcNow.AddDays(-1)) { continue; }
            try { using FileStream abandoned = new(path, FileMode.Open,
FileAccess.ReadWrite, FileShare.None, 1, FileOptions.DeleteOnClose); }
            catch (IOException) { /* Another process still owns or already removed this
stage. */ }
        }
    }
}

```

```

}

/// <summary>
/// Owns a temporary artifact until promotion or cleanup.
/// </summary>
public sealed class StagedArtifact(string stagePath, string blobPath, string identifier, string
fileName, string contentType) : IAsyncDisposable
{
    public string? InstallerIdentityJson { get; internal set; }
    private bool promoted;
    public string StorageIdentifier { get; } = identifier;
    public string FileName { get; } = fileName;
    public string ContentType { get; } = contentType;
    public string Sha256 { get; internal set; } = string.Empty;
    public long SizeBytes { get; internal set; }
    /// <summary>
    /// Moves the validated staged artifact into its permanent archive location.
    /// </summary>
    public void Promote()
    {
        File.Move(stagePath, blobPath, overwrite: false); promoted = true;
    }
    /// <summary>
    /// Asynchronously releases the resources owned by this instance.
    /// </summary>
    public ValueTask DisposeAsync()
    {
        // A promoted blob may have committed even when SQL commit acknowledgement failed.
        Preserve it.
        if (!promoted) { File.Delete(stagePath); }
        return ValueTask.CompletedTask;
    }
}

```

VendorRoleSeeder.cs

```

using System.Security.Cryptography;
using System.Text;
using BridgeControl.Data;
using BridgeControl.Models.Vendor;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;

namespace BridgeControl.Services.Vendor;

/// <summary>
/// Installs the four predefined vendor/customer roles and their permitted actions, without
platform privileges.
/// </summary>
public class VendorRoleSeeder
{
    public const string Administrator = "VendorAdministrator";
    private static readonly string[] AllPermissions = ["users.read", "users.manage",
"customers.read", "customers.manage",
"products.read", "products.manage", "installations.read", "installations.manage",
"configuration.read", "configuration.manage",
"licenses.read", "licenses.manage", "deployments.read", "deployments.manage",
"agents.read", "agents.manage", "logs.read", "monitoring.read"];
    private static readonly Dictionary<string, string[]> Grants = new()
    {

```

```

        [Administrator] = AllPermissions,
        ["VendorTechnician"] = ["customers.read", "products.read", "installations.read",
"installations.manage", "configuration.read", "configuration.manage", "licenses.read",
"deployments.read", "deployments.manage", "agents.read", "logs.read", "monitoring.read"],
        ["CustomerAdministrator"] = ["users.read", "users.manage", "customers.read",
"products.read", "installations.read", "installations.manage", "configuration.read",
"configuration.manage", "licenses.read", "deployments.read", "deployments.manage", "agents.read",
"logs.read", "monitoring.read"],
        ["CustomerUser"] = ["customers.read", "products.read", "installations.read",
"configuration.read", "licenses.read", "deployments.read", "agents.read", "logs.read",
"monitoring.read"]
    };

    /// <summary>
    /// Creates the vendor's built-in roles, permissions, and grants when missing.
    /// </summary>
    public async Task SeedAsync(VendorDbContext database, CancellationToken cancellationToken
= default)
    {
        await using IDbContextTransaction transaction = await
database.Database.BeginTransactionAsync(cancellationToken);
        Dictionary<string, Permission> permissions = await
database.Permissions.ToDictionaryAsync(permission => permission.Code, cancellationToken);
        foreach (string code in AllPermissions)
        {
            if (!permissions.ContainsKey(code))
            {
                Permission permission = new() { Id = SeedId("permission:" + code),
Code = code, Description = code };
                database.Permissions.Add(permission);
                permissions.Add(code, permission);
            }
        }
        List<Role> roles = await database.Roles.Include(role =>
role.RolePermissions).ToListAsync(cancellationToken);
        foreach ((string name, string[] codes) in Grants)
        {
            Role? role = roles.SingleOrDefault(role => role.Name == name);
            if (role is null)
            {
                role = new Role { Id = SeedId("role:" + name), Name = name };
                database.Roles.Add(role);
            }
            role.Scope = name.StartsWith("Vendor", StringComparison.Ordinal) ? "Vendor"
: "Customer";
            HashSet<Guid> allowed = codes.Select(code =>
permissions[code].Id).ToHashSet();
            database.RolePermissions.RemoveRange(role.RolePermissions.Where(grant =>
!allowed.Contains(grant.PermissionId)));
            foreach (Guid permissionId in allowed)
            {
                if (!role.RolePermissions.Any(grant => grant.PermissionId ==
permissionId))
                {
                    role.RolePermissions.Add(new RolePermission
                    {
                        Id = SeedId($"grant:{role.Id}:{permissionId}"),
                        RoleId = role.Id,
                        PermissionId = permissionId
                    });
                }
            }
        }
    }

```

```

    }
    await database.SaveChangesAsync(cancellationToken);
    await transaction.CommitAsync(cancellationToken);
}

/// <summary>
/// Derives a deterministic identifier for a built-in role or permission name.
/// </summary>
private static Guid SeedId(string name) =>
new (SHA256.HashData(Encoding.UTF8.GetBytes("BridgeControl:" + name)).AsSpan(0, 16));
}

```

VendorSecurityAudit.cs

```

using BridgeControl.Data;
using BridgeControl.Models.Vendor;

namespace BridgeControl.Services.Vendor;

/// <summary>Adds safe structured evidence to the caller's business transaction.</summary>
public class VendorSecurityAudit(TimeProvider clock)
{
    /// <summary>
    /// Adds a vendor security audit event to the current database operation.
    /// </summary>
    public void Record(VendorDbContext database, Guid actor, string action, Guid target,
        string outcome = "Succeeded", string entityType = "UserAccount", string actorType
        = "User", Guid? resourceId = null)
    => database.VendorAuditLogs.Add(new VendorAuditLog
    {
        ActorId = actor,
        ActorType = actorType,
        Action = action,
        EntityId = target,
        EntityType = entityType,
        Outcome = outcome,
        Context = resourceId is null ? "{}" :
System.Text.Json.JsonSerializer.Serialize(new { resourceId }),
        CreatedAt = clock.GetUtcNow()
    });
}

```

VendorWorker.cs

```

using BridgeControl.Data;
using Microsoft.EntityFrameworkCore;

namespace BridgeControl.Services.Vendor;

/// <summary>Polls active vendors and resolves a fresh scoped processor for each
vendor.</summary>
public abstract class VendorWorker<TProcessor>(IServiceScopeFactory scopes, Func<bool> enabled,
    Func<int> pollSeconds, ILogger logger, string workerName) : BackgroundService
    where TProcessor : class, IVendorProcessor
{
    /// <summary>Processes bounded vendor pages until the host stops.</summary>
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
    }
}

```

```

        {
            if (!enabled()) { return; }
            Guid? cursor = null;
            while (!stoppingToken.IsCancellationRequested)
            {
                try
                {
                    await Task.Delay(TimeSpan.FromSeconds(pollSeconds()),
stoppingToken);

                    await using AsyncServiceScope scope = scopes.CreateAsyncScope();
                    PlatformDbContext platform =
scope.ServiceProvider.GetRequiredService<PlatformDbContext>();
                    var query = platform.SoftwareVendors.AsNoTracking().Where(x =>
x.Status == "Active");
                    if (cursor is Guid last) { query = query.Where(x =>
x.Id.CompareTo(last) > 0); }
                    Guid[] vendors = await query.OrderBy(x => x.Id).Select(x =>
x.Id).Take(20).ToArrayAsync(stoppingToken);
                    cursor = vendors.Length < 20 ? null : vendors[^1];
                    foreach (Guid vendor in vendors)
                    {
                        try
                        {
                            await using AsyncServiceScope vendorScope =
scopes.CreateAsyncScope();

                            await
vendorScope.ServiceProvider.GetRequiredService<TProcessor>().ProcessVendorAsync(vendor,
stoppingToken);
                        }
                        catch (OperationCanceledException) when
(stoppingToken.IsCancellationRequested) { throw; }
                        catch (Exception error) { logger.LogWarning("{WorkerName}
worker could not process vendor {VendorId}: {ErrorType}. Check migration and database
availability.", workerName, vendor, error.GetType().Name); }
                    }
                }
                catch (OperationCanceledException) when
(stoppingToken.IsCancellationRequested) { break; }
                catch (Exception error) { logger.LogWarning("{WorkerName} worker discovery
failed: {ErrorType}.", workerName, error.GetType().Name); }
            }
        }
    }
}

```

Data/Migrations/Platform

20260915174422_InitialPlatform.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{

```

```

    /// <inheritdoc />
    /// <summary>
    /// Defines the initial platform migration for the platform database.
    /// </summary>
    public partial class InitialPlatform : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the initial platform database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "PlatformAdmins",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    Email = table.Column<string>(type: "nvarchar(254)", maxLength: 254, nullable:
false),
                    PasswordHash = table.Column<string>(type: "nvarchar(512)", maxLength: 512,
nullable: false),
                    Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
                    CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
                    LastLoginAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_PlatformAdmins", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "SoftwareVendors",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    Name = table.Column<string>(type: "nvarchar(200)", maxLength: 200, nullable:
false),
                    Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
                    CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
                    DeactivatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_SoftwareVendors", x => x.Id);
                });

            migrationBuilder.CreateTable(
                name: "PlatformAuditLogs",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    PlatformAdminId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
                    VendorId = table.Column<Guid>(type: "uniqueidentifier", nullable: true),
                    Action = table.Column<string>(type: "nvarchar(100)", maxLength: 100,
nullable: false),

```

```

        EntityType = table.Column<string>(type: "nvarchar(100)", maxLength: 100,
nullable: false),
        EntityId = table.Column<string>(type: "nvarchar(200)", maxLength: 200,
nullable: false),
        Details = table.Column<string>(type: "nvarchar(max)", nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_PlatformAuditLogs", x => x.Id);
        table.ForeignKey(
            name: "FK_PlatformAuditLogs_PlatformAdmins_PlatformAdminId",
            column: x => x.PlatformAdminId,
            principalTable: "PlatformAdmins",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_PlatformAuditLogs_SoftwareVendors_VendorId",
            column: x => x.VendorId,
            principalTable: "SoftwareVendors",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "VendorDatabases",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        VendorId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        DatabaseName = table.Column<string>(type: "nvarchar(128)", maxLength: 128,
nullable: false),
        ConnectionSecretRef = table.Column<string>(type: "nvarchar(512)", maxLength:
512, nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_VendorDatabases", x => x.Id);
        table.ForeignKey(
            name: "FK_VendorDatabases_SoftwareVendors_VendorId",
            column: x => x.VendorId,
            principalTable: "SoftwareVendors",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "VendorFileArchives",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        VendorId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        StorageIdentifier = table.Column<string>(type: "nvarchar(1024)", maxLength:
1024, nullable: false),
        AccessSecretRef = table.Column<string>(type: "nvarchar(512)", maxLength: 512,
nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    }

```

```

    },
    constraints: table =>
    {
        table.PrimaryKey("PK_VendorFileArchives", x => x.Id);
        table.ForeignKey(
            name: "FK_VendorFileArchives_SoftwareVendors_VendorId",
            column: x => x.VendorId,
            principalTable: "SoftwareVendors",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name: "IX_PlatformAdmins_Email",
    table: "PlatformAdmins",
    column: "Email",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_PlatformAuditLogs_PlatformAdminId",
    table: "PlatformAuditLogs",
    column: "PlatformAdminId");

migrationBuilder.CreateIndex(
    name: "IX_PlatformAuditLogs_VendorId",
    table: "PlatformAuditLogs",
    column: "VendorId");

migrationBuilder.CreateIndex(
    name: "IX_VendorDatabases_VendorId",
    table: "VendorDatabases",
    column: "VendorId",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_VendorFileArchives_VendorId",
    table: "VendorFileArchives",
    column: "VendorId",
    unique: true);
}

/// <inheritdoc />
/// <summary>
/// Reverts the initial platform database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "PlatformAuditLogs");

    migrationBuilder.DropTable(
        name: "VendorDatabases");

    migrationBuilder.DropTable(
        name: "VendorFileArchives");

    migrationBuilder.DropTable(
        name: "PlatformAdmins");

    migrationBuilder.DropTable(
        name: "SoftwareVendors");
}
}

```



```
}
```

20260915180313_EnforceVendorNameRequirements.cs

```
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the enforce vendor name requirements migration for the platform database.
    /// </summary>
    public partial class EnforceVendorNameRequirements : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the enforce vendor name requirements database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            // Check existing rows before changing their stored names or adding uniqueness.
            migrationBuilder.Sql("""
                DECLARE @whitespace nvarchar(25) =
                    NCHAR(9) + NCHAR(10) + NCHAR(11) + NCHAR(12) + NCHAR(13) + NCHAR(32) +
                    NCHAR(133) + NCHAR(160) + NCHAR(5760) + NCHAR(8192) + NCHAR(8193) +
NCHAR(8194) +
                    NCHAR(8195) + NCHAR(8196) + NCHAR(8197) + NCHAR(8198) + NCHAR(8199) +
NCHAR(8200) +
                    NCHAR(8201) + NCHAR(8202) + NCHAR(8232) + NCHAR(8233) + NCHAR(8239) +
NCHAR(8287) +
                    NCHAR(12288);

                IF EXISTS (SELECT 1 FROM [SoftwareVendors] WHERE DATALENGTH(TRIM(@whitespace FROM
[Name])) = 0)
                    THROW 51000, 'A vendor name is empty after trimming. Correct it before
applying this migration.', 1;

                IF EXISTS (
                    SELECT 1 FROM [SoftwareVendors]
                    GROUP BY TRIM(@whitespace FROM [Name]) COLLATE Latin1_General_100_CI_AS_SC
                    HAVING COUNT_BIG(*) > 1)
                    THROW 51001, 'Vendor names conflict after trimming and case-insensitive
comparison. Rename the conflicting vendors before applying this migration.', 1;

                UPDATE [SoftwareVendors]
                SET [Name] = TRIM(@whitespace FROM [Name])
                WHERE DATALENGTH([Name]) <> DATALENGTH(TRIM(@whitespace FROM [Name]));
                """);

            migrationBuilder.AlterColumn<string>(
                name: "Name",
                table: "SoftwareVendors",
                type: "nvarchar(200)",
                maxLength: 200,
                nullable: false,
                collation: "Latin1_General_100_CI_AS_SC",
                oldClrType: typeof(string),
                oldType: "nvarchar(200)",
            );
        }
    }
}
```

```

        oldMaxLength: 200);

migrationBuilder.CreateIndex(
    name: "IX_SoftwareVendors_Name",
    table: "SoftwareVendors",
    column: "Name",
    unique: true);

migrationBuilder.AddCheckConstraint(
    name: "CK_SoftwareVendors_Name_TrimmedNotEmpty",
    table: "SoftwareVendors",
    sql: "DATALENGTH([Name]) > 0 AND DATALENGTH(TRIM(NCHAR(9) +
NCHAR(10) + NCHAR(11) + NCHAR(12) + NCHAR(13) + NCHAR(32) + NCHAR(133) + NCHAR(160) + NCHAR(5760)
+ NCHAR(8192) + NCHAR(8193) + NCHAR(8194) + NCHAR(8195) + NCHAR(8196) + NCHAR(8197) + NCHAR(8198)
+ NCHAR(8199) + NCHAR(8200) + NCHAR(8201) + NCHAR(8202) + NCHAR(8232) + NCHAR(8233) + NCHAR(8239)
+ NCHAR(8287) + NCHAR(12288) FROM [Name]))");
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the enforce vendor name requirements database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropIndex(
            name: "IX_SoftwareVendors_Name",
            table: "SoftwareVendors");

        migrationBuilder.DropCheckConstraint(
            name: "CK_SoftwareVendors_Name_TrimmedNotEmpty",
            table: "SoftwareVendors");

        migrationBuilder.AlterColumn<string>(
            name: "Name",
            table: "SoftwareVendors",
            type: "nvarchar(200)",
            maxLength: 200,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(200)",
            oldMaxLength: 200,
            oldCollation: "Latin1_General_100_CI_AS_SC");
    }
}

```

20260916100304_AddPlatformAuthenticationSessions.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add platform authentication sessions migration for the platform database.
    /// </summary>
    public partial class AddPlatformAuthenticationSessions : Migration

```

```

{
    /// <inheritdoc />
    /// <summary>
    /// Applies the add platform authentication sessions database migration.
    /// </summary>
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "AuthenticationSessions",
            columns: table => new
            {
                Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
                AccountId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
                CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: false),
                ExpiresAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: false),
                LastRefreshedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: true),
                RevokedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: true),
                RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_AuthenticationSessions", x => x.Id);
                table.CheckConstraint("CK_AuthenticationSessions_Expiry",
"[ExpiresAt] > [CreatedAt]");
                table.ForeignKey(
                    name:
"FK_AuthenticationSessions_PlatformAdmins_AccountId",
                    column: x => x.AccountId,
                    principalTable: "PlatformAdmins",
                    principalColumn: "Id");
            });

        migrationBuilder.CreateTable(
            name: "RefreshTokens",
            columns: table => new
            {
                Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
                AuthenticationSessionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
                TokenHash = table.Column<byte[]>(type: "varbinary(32)",
maxLength: 32, nullable: false),
                CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: false),
                ExpiresAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: false),
                UsedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset(7)", precision: 7, nullable: true),
                RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_RefreshTokens", x => x.Id);
            });
    }
}

```

```

        table.CheckConstraint("CK_RefreshTokens_Expiry",
"[ExpiresAt] > [CreatedAt]");
        table.CheckConstraint("CK_RefreshTokens_TokenHash",
"DATALENGTH([TokenHash]) = 32");
        table.ForeignKey(
            name:
"FK_RefreshTokens_AuthenticationSessions_AuthenticationSessionId",
            column: x => x.AuthenticationSessionId,
            principalTable: "AuthenticationSessions",
            principalColumn: "Id");
    });

    migrationBuilder.CreateIndex(
        name: "IX_AuthenticationSessions_AccountId",
        table: "AuthenticationSessions",
        column: "AccountId");

    migrationBuilder.CreateIndex(
        name: "IX_RefreshTokens_AuthenticationSessionId",
        table: "RefreshTokens",
        column: "AuthenticationSessionId");

    migrationBuilder.CreateIndex(
        name: "IX_RefreshTokens_TokenHash",
        table: "RefreshTokens",
        column: "TokenHash",
        unique: true);
}

/// <inheritdoc />
/// <summary>
/// Reverts the add platform authentication sessions database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "RefreshTokens");

    migrationBuilder.DropTable(
        name: "AuthenticationSessions");
}
}

```

20260916145018_AddVendorProvisioning.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add vendor provisioning migration for the platform database.
    /// </summary>
    public partial class AddVendorProvisioning : Migration
    {
        /// <inheritdoc />

```

```

    /// <summary>
    /// Applies the add vendor provisioning database migration.
    /// </summary>
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<byte[]>(
            name: "RowVersion",
            table: "SoftwareVendors",
            type: "rowversion",
            rowVersion: true,
            nullable: false,
            defaultValue: new byte[0]);

        migrationBuilder.CreateTable(
            name: "VendorProvisioning",
            columns: table => new
            {
                VendorId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                DatabaseReadyAt = table.Column<DateTimeOffset>(type: "datetimeoffset",
nullable: true),
                ArchiveReadyAt = table.Column<DateTimeOffset>(type: "datetimeoffset",
nullable: true),
                SeededAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
true),
                InvitationEstablishedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
                SchemaVersion = table.Column<string>(type: "nvarchar(150)", maxLength: 150,
nullable: false),
                FailureCode = table.Column<string>(type: "nvarchar(64)", maxLength: 64,
nullable: true),
                UpdatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
false)

            },
            constraints: table =>
            {
                table.PrimaryKey("PK_VendorProvisioning", x => x.VendorId);
                table.ForeignKey(
                    name: "FK_VendorProvisioning_SoftwareVendors_VendorId",
                    column: x => x.VendorId,
                    principalTable: "SoftwareVendors",
                    principalColumn: "Id");
            });
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add vendor provisioning database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "VendorProvisioning");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "SoftwareVendors");
    }
}

```

20260916154121_AddVendorSessionRevocationCutoff.cs

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add vendor session revocation cutoff migration for the platform database.
    /// </summary>
    public partial class AddVendorSessionRevocationCutoff : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add vendor session revocation cutoff database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<DateTimeOffset>(
                name: "SessionsRevokedBefore",
                table: "SoftwareVendors",
                type: "datetimeoffset",
                nullable: true);
        }

        /// <inheritdoc />
        /// <summary>
        /// Reverts the add vendor session revocation cutoff database migration.
        /// </summary>
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropColumn(
                name: "SessionsRevokedBefore",
                table: "SoftwareVendors");
        }
    }
}
```

20260917101020_AddVendorMaintenanceOperations.cs

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Platform
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add vendor maintenance operations migration for the platform database.
    /// </summary>
    public partial class AddVendorMaintenanceOperations : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add vendor maintenance operations database migration.
    }
}
```

```

/// </summary>
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "VendorMaintenanceOperations",
        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
            VendorId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
            PlatformAdminId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
            Kind = table.Column<string>(type: "nvarchar(16)", maxLength:
16, nullable: false),
            Status = table.Column<string>(type: "nvarchar(16)",
maxLength: 16, nullable: false),
            IsActive = table.Column<bool>(type: "bit", nullable: false),
            Stage = table.Column<string>(type: "nvarchar(150)",
maxLength: 150, nullable: false),
            SourceSchema = table.Column<string>(type: "nvarchar(150)",
maxLength: 150, nullable: false),
            CurrentSchema = table.Column<string>(type: "nvarchar(150)",
maxLength: 150, nullable: false),
            TargetSchema = table.Column<string>(type: "nvarchar(150)",
maxLength: 150, nullable: false),
            BackupId = table.Column<Guid>(type: "uniqueidentifier",
nullable: true),
            BundleSha256 = table.Column<string>(type: "nvarchar(64)",
maxLength: 64, nullable: true),
            FailureCode = table.Column<string>(type: "nvarchar(64)",
maxLength: 64, nullable: true),
            OperatorAccount = table.Column<string>(type:
"nvarchar(256)", maxLength: 256, nullable: false),
            OperatorHost = table.Column<string>(type: "nvarchar(256)",
maxLength: 256, nullable: false),
            CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
            UpdatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
            CompletedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
            ExpiredAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
            RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_VendorMaintenanceOperations", x =>
x.Id);
            table.ForeignKey(
                name:
"FK_VendorMaintenanceOperations_PlatformAdmins_PlatformAdminId",
                column: x => x.PlatformAdminId,
                principalTable: "PlatformAdmins",
                principalColumn: "Id");
            table.ForeignKey(
                name:
"FK_VendorMaintenanceOperations_SoftwareVendors_VendorId",
                column: x => x.VendorId,
                principalTable: "SoftwareVendors",

```

```

                principalColumn: "Id");
        });

        migrationBuilder.CreateIndex(
            name: "IX_VendorMaintenanceOperations_Kind_Status_CompletedAt",
            table: "VendorMaintenanceOperations",
            columns: new[] { "Kind", "Status", "CompletedAt" });

        migrationBuilder.CreateIndex(
            name: "IX_VendorMaintenanceOperations_PlatformAdminId",
            table: "VendorMaintenanceOperations",
            column: "PlatformAdminId");

        migrationBuilder.CreateIndex(
            name: "IX_VendorMaintenanceOperations_VendorId",
            table: "VendorMaintenanceOperations",
            column: "VendorId",
            unique: true,
            filter: "[IsActive] = 1");
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add vendor maintenance operations database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "VendorMaintenanceOperations");
    }
}

```

Data/Migrations/Vendor

20260915174441_InitialVendor.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the initial vendor migration for the vendor database.
    /// </summary>
    public partial class InitialVendor : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the initial vendor database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {

```



```

migrationBuilder.CreateTable(
    name: "Customers",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        Name = table.Column<string>(type: "nvarchar(200)", maxLength: 200, nullable:
false),
        Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
        DeactivatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Customers", x => x.Id);
    });

migrationBuilder.CreateTable(
    name: "FileArtifacts",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ArtifactType = table.Column<string>(type: "nvarchar(32)", maxLength: 32,
nullable: false),
        FileName = table.Column<string>(type: "nvarchar(255)", maxLength: 255,
nullable: false),
        StorageIdentifier = table.Column<string>(type: "nvarchar(1024)", maxLength:
1024, nullable: false),
        ContentType = table.Column<string>(type: "nvarchar(255)", maxLength: 255,
nullable: false),
        Sha256 = table.Column<string>(type: "varchar(64)", unicode: false, maxLength:
64, nullable: false),
        SizeBytes = table.Column<long>(type: "bigint", nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_FileArtifacts", x => x.Id);
        table.CheckConstraint("CK_FileArtifacts_Sha256", "DATALENGTH([Sha256]) = 64
AND [Sha256] COLLATE Latin1_General_100_BIN2 NOT LIKE '%[^0-9A-Fa-f]%'");
    });

migrationBuilder.CreateTable(
    name: "Permissions",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        Code = table.Column<string>(type: "nvarchar(100)", maxLength: 100, nullable:
false),
        Description = table.Column<string>(type: "nvarchar(2000)", maxLength: 2000,
nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Permissions", x => x.Id);
    });

migrationBuilder.CreateTable(
    name: "Products",

```

```

        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
            Name = table.Column<string>(type: "nvarchar(200)", maxLength: 200, nullable:
false),
            Description = table.Column<string>(type: "nvarchar(2000)", maxLength: 2000,
nullable: false),
            Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
            CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Products", x => x.Id);
        });

migrationBuilder.CreateTable(
    name: "Roles",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        Name = table.Column<string>(type: "nvarchar(200)", maxLength: 200, nullable:
false),
        Scope = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Roles", x => x.Id);
    });

migrationBuilder.CreateTable(
    name: "UserAccounts",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier", nullable: true),
        Email = table.Column<string>(type: "nvarchar(254)", maxLength: 254, nullable:
false),
        PasswordHash = table.Column<string>(type: "nvarchar(512)", maxLength: 512,
nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
        LastLoginAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_UserAccounts", x => x.Id);
        table.ForeignKey(
            name: "FK_UserAccounts_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "ProductVersions",
    columns: table => new

```

```

        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
            ProductId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
            Version = table.Column<string>(type: "nvarchar(100)", maxLength: 100,
nullable: false),
            Status = table.Column<string>(type: "nvarchar(32)", maxLength: 32, nullable:
false),
            CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
            PublishedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_ProductVersions", x => x.Id);
            table.UniqueConstraint("AK_ProductVersions_Id_ProductId", x => new { x.Id,
x.ProductId });
            table.ForeignKey(
                name: "FK_ProductVersions_Products_ProductId",
                column: x => x.ProductId,
                principalTable: "Products",
                principalColumn: "Id");
        });

migrationBuilder.CreateTable(
    name: "RolePermissions",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        RoleId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        PermissionId = table.Column<Guid>(type: "uniqueidentifier", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_RolePermissions", x => x.Id);
        table.ForeignKey(
            name: "FK_RolePermissions_Permissions_PermissionId",
            column: x => x.PermissionId,
            principalTable: "Permissions",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_RolePermissions_Roles_RoleId",
            column: x => x.RoleId,
            principalTable: "Roles",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "TechnicianCustomerAccessGrants",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        UserId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        GrantedByUserId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        GrantedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_TechnicianCustomerAccessGrants", x => x.Id);
    });

```

```

        table.ForeignKey(
            name: "FK_TechnicianCustomerAccessGrants_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_TechnicianCustomerAccessGrants_UserAccounts_GrantedByUserId",
            column: x => x.GrantedByUserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_TechnicianCustomerAccessGrants_UserAccounts_UserId",
            column: x => x.UserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "UserRoles",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        UserId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        RoleId = table.Column<Guid>(type: "uniqueidentifier", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_UserRoles", x => x.Id);
        table.ForeignKey(
            name: "FK_UserRoles_Roles_RoleId",
            column: x => x.RoleId,
            principalTable: "Roles",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_UserRoles_UserAccounts_UserId",
            column: x => x.UserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "ConfigurationMigrations",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        FromProductVersionId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        ToProductVersionId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        CreatedByUserId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ConfigurationMigrations", x => x.Id);
        table.ForeignKey(
            name: "FK_ConfigurationMigrations_ProductVersions_FromProductVersionId",
            column: x => x.FromProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
    });

```

```

        table.ForeignKey(
            name: "FK_ConfigurationMigrations_ProductVersions_ToProductVersionId",
            column: x => x.ToProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_ConfigurationMigrations_UserAccounts_CreatedByUserId",
            column: x => x.CreatedByUserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "ProductVersionArtifacts",
    columns: table => new
    {
        ProductVersionId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        FileArtifactId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ProductVersionArtifacts", x => new { x.ProductVersionId,
x.FileArtifactId });
        table.ForeignKey(
            name: "FK_ProductVersionArtifacts_FileArtifacts_FileArtifactId",
            column: x => x.FileArtifactId,
            principalTable: "FileArtifacts",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_ProductVersionArtifacts_ProductVersions_ProductVersionId",
            column: x => x.ProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "MigrationRules",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        MigrationId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        Sequence = table.Column<int>(type: "int", nullable: false),
        RuleType = table.Column<string>(type: "nvarchar(64)", maxLength: 64,
nullable: false),
        SourcePath = table.Column<string>(type: "nvarchar(2048)", maxLength: 2048,
nullable: false),
        TargetPath = table.Column<string>(type: "nvarchar(2048)", maxLength: 2048,
nullable: false),
        Transform = table.Column<string>(type: "nvarchar(max)", nullable: false),
        IsDestructive = table.Column<bool>(type: "bit", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_MigrationRules", x => x.Id);
        table.ForeignKey(
            name: "FK_MigrationRules_ConfigurationMigrations_MigrationId",
            column: x => x.MigrationId,
            principalTable: "ConfigurationMigrations",
            principalColumn: "Id");
    });

```

```

migrationBuilder.CreateTable(
    name: "ConfigurationProfiles",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ProductId = table.Column<Guid>(type: "uniqueidentifier", nullable: true),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier", nullable: true),
        InstallationId = table.Column<Guid>(type: "uniqueidentifier", nullable:
true),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ConfigurationProfiles", x => x.Id);
        table.CheckConstraint("CK_ConfigurationProfiles_Scope", "([ProductId] IS NOT
NULL AND [InstallationId] IS NULL) OR ([InstallationId] IS NOT NULL AND [ProductId] IS NULL AND
[CustomerId] IS NULL)");
        table.ForeignKey(
            name: "FK_ConfigurationProfiles_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_ConfigurationProfiles_Products_ProductId",
            column: x => x.ProductId,
            principalTable: "Products",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "ConfigurationRevisions",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ConfigurationProfileId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        ProductVersionId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        FileArtifactId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        RevisionNumber = table.Column<int>(type: "int", nullable: false),
        CreatedByUserId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ConfigurationRevisions", x => x.Id);
        table.ForeignKey(
            name:
"FK_ConfigurationRevisions_ConfigurationProfiles_ConfigurationProfileId",
            column: x => x.ConfigurationProfileId,
            principalTable: "ConfigurationProfiles",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_ConfigurationRevisions_FileArtifacts_FileArtifactId",
            column: x => x.FileArtifactId,
            principalTable: "FileArtifacts",
            principalColumn: "Id");
    });

```

```

        table.ForeignKey(
            name: "FK_ConfigurationRevisions_ProductVersions_ProductVersionId",
            column: x => x.ProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_ConfigurationRevisions_UserAccounts_CreatedByUserId",
            column: x => x.CreatedByUserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "EffectiveConfigurationRevisions",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        InstallationId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        ProductId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ProductVersionId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        FileArtifactId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        RevisionNumber = table.Column<int>(type: "int", nullable: false),
        CreatedByUserId = table.Column<Guid>(type: "uniqueidentifier", nullable:
true),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_EffectiveConfigurationRevisions", x => x.Id);

table.UniqueConstraint("AK_EffectiveConfigurationRevisions_Id_InstallationId", x => new { x.Id,
x.InstallationId });
        table.ForeignKey(
            name: "FK_EffectiveConfigurationRevisions_FileArtifacts_FileArtifactId",
            column: x => x.FileArtifactId,
            principalTable: "FileArtifacts",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_EffectiveConfigurationRevisions_ProductVersions_ProductVersionId_ProductId",
            columns: x => new { x.ProductVersionId, x.ProductId },
            principalTable: "ProductVersions",
            principalColumns: new[] { "Id", "ProductId" });
        table.ForeignKey(
            name: "FK_EffectiveConfigurationRevisions_UserAccounts_CreatedByUserId",
            column: x => x.CreatedByUserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "EffectiveConfigurationSourceRevisions",
    columns: table => new
    {
        EffectiveConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        ConfigurationRevisionId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false)
    }

```

```

    },
    constraints: table =>
    {
        table.PrimaryKey("PK_EffectiveConfigurationSourceRevisions", x => new {
x.EffectiveConfigurationRevisionId, x.ConfigurationRevisionId });
        table.ForeignKey(
            name:
"FK_EffectiveConfigurationSourceRevisions_ConfigurationRevisions_ConfigurationRevisionId",
            column: x => x.ConfigurationRevisionId,
            principalTable: "ConfigurationRevisions",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_EffectiveConfigurationSourceRevisions_EffectiveConfigurationRevisions_EffectiveConfigurationRevisionId",
            column: x => x.EffectiveConfigurationRevisionId,
            principalTable: "EffectiveConfigurationRevisions",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "Installations",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ProductId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CurrentEffectiveConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
        Name = table.Column<string>(type: "nvarchar(200)", maxLength: 200, nullable:
false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Installations", x => x.Id);
        table.UniqueConstraint("AK_Installations_Id_ProductId", x => new { x.Id,
x.ProductId });
        table.ForeignKey(
            name: "FK_Installations_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_Installations_EffectiveConfigurationRevisions_CurrentEffectiveConfigurationRevisionId_Id",
            columns: x => new { x.CurrentEffectiveConfigurationRevisionId, x.Id },
            principalTable: "EffectiveConfigurationRevisions",
            principalColumns: new[] { "Id", "InstallationId" });
        table.ForeignKey(
            name: "FK_Installations_Products_ProductId",
            column: x => x.ProductId,
            principalTable: "Products",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationMigrations_CreatedByUserId",
    table: "ConfigurationMigrations",
    column: "CreatedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationMigrations_FromProductVersionId",

```



```

        table: "ConfigurationMigrations",
        column: "FromProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationMigrations_ToProductVersionId",
    table: "ConfigurationMigrations",
    column: "ToProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationProfiles_CustomerId_ProductId",
    table: "ConfigurationProfiles",
    columns: new[] { "CustomerId", "ProductId" },
    unique: true,
    filter: "[CustomerId] IS NOT NULL AND [ProductId] IS NOT NULL AND
[InstallationId] IS NULL");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationProfiles_InstallationId",
    table: "ConfigurationProfiles",
    column: "InstallationId",
    unique: true,
    filter: "[InstallationId] IS NOT NULL");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationProfiles_ProductId",
    table: "ConfigurationProfiles",
    column: "ProductId",
    unique: true,
    filter: "[ProductId] IS NOT NULL AND [CustomerId] IS NULL AND [InstallationId] IS
NULL");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationRevisions_ConfigurationProfileId_RevisionNumber",
    table: "ConfigurationRevisions",
    columns: new[] { "ConfigurationProfileId", "RevisionNumber" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationRevisions_CreatedByUserId",
    table: "ConfigurationRevisions",
    column: "CreatedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationRevisions_FileArtifactId",
    table: "ConfigurationRevisions",
    column: "FileArtifactId");

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationRevisions_ProductVersionId",
    table: "ConfigurationRevisions",
    column: "ProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_CreatedByUserId",
    table: "EffectiveConfigurationRevisions",
    column: "CreatedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_FileArtifactId",
    table: "EffectiveConfigurationRevisions",
    column: "FileArtifactId");

```

```

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_InstallationId_ProductId",
    table: "EffectiveConfigurationRevisions",
    columns: new[] { "InstallationId", "ProductId" });

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_InstallationId_RevisionNumber",
    table: "EffectiveConfigurationRevisions",
    columns: new[] { "InstallationId", "RevisionNumber" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_ProductVersionId_ProductId",
    table: "EffectiveConfigurationRevisions",
    columns: new[] { "ProductVersionId", "ProductId" });

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationSourceRevisions_ConfigurationRevisionId",
    table: "EffectiveConfigurationSourceRevisions",
    column: "ConfigurationRevisionId");

migrationBuilder.CreateIndex(
    name: "IX_Installations_CurrentEffectiveConfigurationRevisionId_Id",
    table: "Installations",
    columns: new[] { "CurrentEffectiveConfigurationRevisionId", "Id" });

migrationBuilder.CreateIndex(
    name: "IX_Installations_CustomerId",
    table: "Installations",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_Installations_ProductId",
    table: "Installations",
    column: "ProductId");

migrationBuilder.CreateIndex(
    name: "IX_MigrationRules_MigrationId_Sequence",
    table: "MigrationRules",
    columns: new[] { "MigrationId", "Sequence" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_Permissions_Code",
    table: "Permissions",
    column: "Code",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_ProductVersionArtifacts_FileArtifactId",
    table: "ProductVersionArtifacts",
    column: "FileArtifactId");

migrationBuilder.CreateIndex(
    name: "IX_ProductVersions_ProductId_Version",
    table: "ProductVersions",
    columns: new[] { "ProductId", "Version" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_RolePermissions_PermissionId",
    table: "RolePermissions",

```

```

        column: "PermissionId");

migrationBuilder.CreateIndex(
    name: "IX_RolePermissions_RoleId_PermissionId",
    table: "RolePermissions",
    columns: new[] { "RoleId", "PermissionId" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_Roles_Name",
    table: "Roles",
    column: "Name",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_TechnicianCustomerAccessGrants_CustomerId",
    table: "TechnicianCustomerAccessGrants",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_TechnicianCustomerAccessGrants_GrantedByUserId",
    table: "TechnicianCustomerAccessGrants",
    column: "GrantedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_TechnicianCustomerAccessGrants_UserId_CustomerId",
    table: "TechnicianCustomerAccessGrants",
    columns: new[] { "UserId", "CustomerId" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_UserAccounts_CustomerId",
    table: "UserAccounts",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_UserAccounts_Email",
    table: "UserAccounts",
    column: "Email",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_UserRoles_RoleId",
    table: "UserRoles",
    column: "RoleId");

migrationBuilder.CreateIndex(
    name: "IX_UserRoles_UserId_RoleId",
    table: "UserRoles",
    columns: new[] { "UserId", "RoleId" },
    unique: true);

migrationBuilder.AddForeignKey(
    name: "FK_ConfigurationProfiles_Installations_InstallationId",
    table: "ConfigurationProfiles",
    column: "InstallationId",
    principalTable: "Installations",
    principalColumn: "Id");

migrationBuilder.AddForeignKey(
    name:
"FK_EffectiveConfigurationRevisions_Installations_InstallationId_ProductId",

```

```

        table: "EffectiveConfigurationRevisions",
        columns: new[] { "InstallationId", "ProductId" },
        principalTable: "Installations",
        principalColumns: new[] { "Id", "ProductId" });
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the initial vendor database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropForeignKey(
            name:
"FK_EffectiveConfigurationRevisions_ProductVersions_ProductVersionId_ProductId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropForeignKey(
            name: "FK_EffectiveConfigurationRevisions_UserAccounts_CreatedByUserId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropForeignKey(
            name: "FK_Installations_Customers_CustomerId",
            table: "Installations");

        migrationBuilder.DropForeignKey(
            name:
"FK_EffectiveConfigurationRevisions_Installations_InstallationId_ProductId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropTable(
            name: "EffectiveConfigurationSourceRevisions");

        migrationBuilder.DropTable(
            name: "MigrationRules");

        migrationBuilder.DropTable(
            name: "ProductVersionArtifacts");

        migrationBuilder.DropTable(
            name: "RolePermissions");

        migrationBuilder.DropTable(
            name: "TechnicianCustomerAccessGrants");

        migrationBuilder.DropTable(
            name: "UserRoles");

        migrationBuilder.DropTable(
            name: "ConfigurationRevisions");

        migrationBuilder.DropTable(
            name: "ConfigurationMigrations");

        migrationBuilder.DropTable(
            name: "Permissions");

        migrationBuilder.DropTable(
            name: "Roles");

        migrationBuilder.DropTable(
            name: "ConfigurationProfiles");
    }

```

```

        migrationBuilder.DropTable(
            name: "ProductVersions");

        migrationBuilder.DropTable(
            name: "UserAccounts");

        migrationBuilder.DropTable(
            name: "Customers");

        migrationBuilder.DropTable(
            name: "Installations");

        migrationBuilder.DropTable(
            name: "EffectiveConfigurationRevisions");

        migrationBuilder.DropTable(
            name: "Products");

        migrationBuilder.DropTable(
            name: "FileArtifacts");
    }
}

```

20260916143834_AddVendorAuthenticationSessions.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add vendor authentication sessions migration for the vendor database.
    /// </summary>
    public partial class AddVendorAuthenticationSessions : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add vendor authentication sessions database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "AuthenticationSessions",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    AccountId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
                    ExpiresAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
                    LastRefreshedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true),
                    RevokedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: true),

```

```

        RowVersion = table.Column<byte[]>(type: "rowversion", rowVersion: true,
nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AuthenticationSessions", x => x.Id);
        table.CheckConstraint("CK_AuthenticationSessions_Expiry", "[ExpiresAt] >
[CreatedAt]");

        table.ForeignKey(
            name: "FK_AuthenticationSessions_UserAccounts_AccountId",
            column: x => x.AccountId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "RefreshTokens",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        AuthenticationSessionId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        TokenHash = table.Column<byte[]>(type: "varbinary(32)", maxLength: 32,
nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
        ExpiresAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)",
precision: 7, nullable: false),
        UsedAt = table.Column<DateTimeOffset>(type: "datetimeoffset(7)", precision:
7, nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion", rowVersion: true,
nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_RefreshTokens", x => x.Id);
        table.CheckConstraint("CK_RefreshTokens_Expiry", "[ExpiresAt] >
[CreatedAt]");

        table.CheckConstraint("CK_RefreshTokens_TokenHash", "DATALENGTH([TokenHash])
= 32");

        table.ForeignKey(
            name: "FK_RefreshTokens_AuthenticationSessions_AuthenticationSessionId",
            column: x => x.AuthenticationSessionId,
            principalTable: "AuthenticationSessions",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name: "IX_AuthenticationSessions_AccountId",
    table: "AuthenticationSessions",
    column: "AccountId");

migrationBuilder.CreateIndex(
    name: "IX_RefreshTokens_AuthenticationSessionId",
    table: "RefreshTokens",
    column: "AuthenticationSessionId");

migrationBuilder.CreateIndex(
    name: "IX_RefreshTokens_TokenHash",
    table: "RefreshTokens",
    column: "TokenHash",
    unique: true);

```

```

    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add vendor authentication sessions database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "RefreshTokens");

        migrationBuilder.DropTable(
            name: "AuthenticationSessions");
    }
}

```

20260916145033_AddVendorAccessAndInvitations.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add vendor access and invitations migration for the vendor database.
    /// </summary>
    public partial class AddVendorAccessAndInvitations : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add vendor access and invitations database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AlterColumn<string>(
                name: "Email",
                table: "UserAccounts",
                type: "nvarchar(254)",
                maxLength: 254,
                nullable: false,
                collation: "Latin1_General_100_CI_AS_SC",
                oldClrType: typeof(string),
                oldType: "nvarchar(254)",
                oldMaxLength: 254);

            migrationBuilder.AddUniqueConstraint(
                name: "AK_Installations_Id_CustomerId",
                table: "Installations",
                columns: new[] { "Id", "CustomerId" });

            migrationBuilder.CreateTable(
                name: "AccountActivations",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
                    UserId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),

```

```

        TokenHash = table.Column<byte[]>(type: "binary(32)", fixedLength: true,
maxLength: 32, nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
false),
        ExpiresAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
false),
        UsedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
true),
        RevokedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
true),
        DeliveredAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
true),
        ProtectedToken = table.Column<string>(type: "nvarchar(2048)", maxLength:
2048, nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion", rowVersion: true,
nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AccountActivations", x => x.Id);
        table.CheckConstraint("CK_AccountActivations_Expiry", "[ExpiresAt] >
[CreatedAt]");
        table.ForeignKey(
            name: "FK_AccountActivations_UserAccounts_UserId",
            column: x => x.UserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "CustomerUserInstallationAccessGrants",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        UserId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        InstallationId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        GrantedByUserId = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        GrantedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_CustomerUserInstallationAccessGrants", x => x.Id);
        table.ForeignKey(
            name:
"FK_CustomerUserInstallationAccessGrants_Installations_InstallationId_CustomerId",
            columns: x => new { x.InstallationId, x.CustomerId },
            principalTable: "Installations",
            principalColumns: new[] { "Id", "CustomerId" });
        table.ForeignKey(
            name:
"FK_CustomerUserInstallationAccessGrants_UserAccounts_GrantedByUserId",
            column: x => x.GrantedByUserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_CustomerUserInstallationAccessGrants_UserAccounts_UserId",
            column: x => x.UserId,
            principalTable: "UserAccounts",

```



```

        principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "VendorAuditLogs",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ActorId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        ActorType = table.Column<string>(type: "nvarchar(32)", maxLength: 32,
nullable: false),
        Action = table.Column<string>(type: "nvarchar(100)", maxLength: 100,
nullable: false),
        EntityId = table.Column<Guid>(type: "uniqueidentifier", nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type: "datetimeoffset", nullable:
false)

    },
    constraints: table =>
    {
        table.PrimaryKey("PK_VendorAuditLogs", x => x.Id);
    });

migrationBuilder.CreateIndex(
    name: "IX_UserAccounts_Id_CustomerId",
    table: "UserAccounts",
    columns: new[] { "Id", "CustomerId" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_AccountActivations_UserId",
    table: "AccountActivations",
    column: "UserId",
    unique: true,
    filter: "[UsedAt] IS NULL AND [RevokedAt] IS NULL");

migrationBuilder.CreateIndex(
    name: "IX_CustomerUserInstallationAccessGrants_GrantedByUserId",
    table: "CustomerUserInstallationAccessGrants",
    column: "GrantedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_CustomerUserInstallationAccessGrants_InstallationId_CustomerId",
    table: "CustomerUserInstallationAccessGrants",
    columns: new[] { "InstallationId", "CustomerId" });

migrationBuilder.CreateIndex(
    name: "IX_CustomerUserInstallationAccessGrants_UserId_InstallationId",
    table: "CustomerUserInstallationAccessGrants",
    columns: new[] { "UserId", "InstallationId" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_VendorAuditLogs_EntityId_CreatedAt",
    table: "VendorAuditLogs",
    columns: new[] { "EntityId", "CreatedAt" });

// Keep vendor-level UserAccount.CustomerId nullable while enforcing customer
ownership in SQL.
migrationBuilder.AddForeignKey(
    name: "FK_CustomerInstallationAccess_UserCustomer",
    table: "CustomerUserInstallationAccessGrants",
    columns: new[] { "UserId", "CustomerId" },

```

```

        principalTable: "UserAccounts",
        principalColumns: new[] { "Id", "CustomerId" });
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add vendor access and invitations database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "AccountActivations");

        migrationBuilder.DropTable(
            name: "CustomerUserInstallationAccessGrants");

        migrationBuilder.DropTable(
            name: "VendorAuditLogs");

        migrationBuilder.DropIndex(
            name: "IX_UserAccounts_Id_CustomerId",
            table: "UserAccounts");

        migrationBuilder.DropUniqueConstraint(
            name: "AK_Installations_Id_CustomerId",
            table: "Installations");

        migrationBuilder.AlterColumn<string>(
            name: "Email",
            table: "UserAccounts",
            type: "nvarchar(254)",
            maxLength: 254,
            nullable: false,
            oldClrType: typeof(string),
            oldType: "nvarchar(254)",
            oldMaxLength: 254,
            oldCollation: "Latin1_General_100_CI_AS_SC");
    }
}

```

20260916154053_AddTenantRecoveryAndSecurityAudit.c

S

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add tenant recovery and security audit migration for the vendor database.
    /// </summary>
    public partial class AddTenantRecoveryAndSecurityAudit : Migration
    {
        /// <inheritdoc />

```

```

/// <summary>
/// Applies the add tenant recovery and security audit database migration.
/// </summary>
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "Context",
        table: "VendorAuditLogs",
        type: "nvarchar(512)",
        maxLength: 512,
        nullable: false,
        defaultValue: "{}");

    migrationBuilder.AddColumn<string>(
        name: "EntityType",
        table: "VendorAuditLogs",
        type: "nvarchar(64)",
        maxLength: 64,
        nullable: false,
        defaultValue: "UserAccount");

    migrationBuilder.AddColumn<string>(
        name: "Outcome",
        table: "VendorAuditLogs",
        type: "nvarchar(32)",
        maxLength: 32,
        nullable: false,
        defaultValue: "Succeeded");

    migrationBuilder.CreateTable(
        name: "PasswordResets",
        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
            UserId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
            TokenHash = table.Column<byte[]>(type: "binary(32)",
fixedLength: true, maxLength: 32, nullable: false),
            CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
            ExpiresAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
            UsedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
            RevokedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
            DeliveredAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
            ProtectedToken = table.Column<string>(type:
"nvarchar(2048)", maxLength: 2048, nullable: true),
            RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_PasswordResets", x => x.Id);
            table.CheckConstraint("CK_PasswordResets_Expiry",
"[ExpiresAt] > [CreatedAt]");

            table.ForeignKey(
                name: "FK_PasswordResets_UserAccounts_UserId",
                column: x => x.UserId,

```

```

        principalTable: "UserAccounts",
        principalColumn: "Id");
    });

    migrationBuilder.CreateIndex(
        name: "IX_PasswordResets_UserId",
        table: "PasswordResets",
        column: "UserId",
        unique: true,
        filter: "[UsedAt] IS NULL AND [RevokedAt] IS NULL");
}

/// <inheritdoc />
/// <summary>
/// Reverts the add tenant recovery and security audit database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "PasswordResets");

    migrationBuilder.DropColumn(
        name: "Context",
        table: "VendorAuditLogs");

    migrationBuilder.DropColumn(
        name: "EntityType",
        table: "VendorAuditLogs");

    migrationBuilder.DropColumn(
        name: "Outcome",
        table: "VendorAuditLogs");
}
}
}

```

20260916161929_AddUserCustomerConcurrency.cs

```

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add user customer concurrency migration for the vendor database.
    /// </summary>
    public partial class AddUserCustomerConcurrency : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add user customer concurrency database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<byte[]>(
                name: "RowVersion",
                table: "UserAccounts",
                type: "rowversion",

```

```

        rowVersion: true,
        nullable: false,
        defaultValue: new byte[0]);

    migrationBuilder.AddColumn<byte[]>(
        name: "RowVersion",
        table: "Customers",
        type: "rowversion",
        rowVersion: true,
        nullable: false,
        defaultValue: new byte[0]);
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add user customer concurrency database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "UserAccounts");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "Customers");
    }
}

```

20260916181224_AddProductArtifactAdministration.cs

```

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add product artifact administration migration for the vendor database.
    /// </summary>
    public partial class AddProductArtifactAdministration : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add product artifact administration database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropForeignKey(
                name: "FK_ProductVersionArtifacts_FileArtifacts_FileArtifactId",
                table: "ProductVersionArtifacts");

            migrationBuilder.DropIndex(
                name: "IX_ProductVersionArtifacts_FileArtifactId",
                table: "ProductVersionArtifacts");

            migrationBuilder.AddColumn<byte[]>(
                name: "RowVersion",

```

```

        table: "ProductVersions",
        type: "rowversion",
        rowVersion: true,
        nullable: false,
        defaultValue: new byte[0]);

migrationBuilder.AddColumn<string>(
    name: "ArtifactType",
    table: "ProductVersionArtifacts",
    type: "nvarchar(32)",
    maxLength: 32,
    nullable: false,
    defaultValue: "");

migrationBuilder.AlterColumn<string>(
    name: "Description",
    table: "Products",
    type: "nvarchar(2000)",
    maxLength: 2000,
    nullable: true,
    oldClrType: typeof(string),
    oldType: "nvarchar(2000)",
    oldMaxLength: 2000);

migrationBuilder.AddColumn<byte[]>(
    name: "RowVersion",
    table: "Products",
    type: "rowversion",
    rowVersion: true,
    nullable: false,
    defaultValue: new byte[0]);

// Preserve the kind of pre-existing associations; ambiguous duplicates
fail the unique index safely.
migrationBuilder.Sql("UPDATE p SET ArtifactType = f.ArtifactType FROM
ProductVersionArtifacts p JOIN FileArtifacts f ON f.Id = p.FileArtifactId;");

migrationBuilder.AddUniqueConstraint(
    name: "AK_FileArtifacts_Id_ArtifactType",
    table: "FileArtifacts",
    columns: new[] { "Id", "ArtifactType" });

migrationBuilder.CreateIndex(
    name: "IX_ProductVersionArtifacts_FileArtifactId_ArtifactType",
    table: "ProductVersionArtifacts",
    columns: new[] { "FileArtifactId", "ArtifactType" });

migrationBuilder.CreateIndex(
    name: "IX_ProductVersionArtifacts_ProductVersionId_ArtifactType",
    table: "ProductVersionArtifacts",
    columns: new[] { "ProductVersionId", "ArtifactType" },
    unique: true);

migrationBuilder.AddForeignKey(
    name:
"FK_ProductVersionArtifacts_FileArtifacts_FileArtifactId_ArtifactType",
    table: "ProductVersionArtifacts",
    columns: new[] { "FileArtifactId", "ArtifactType" },
    principalTable: "FileArtifacts",
    principalColumns: new[] { "Id", "ArtifactType" });
}

```

```

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add product artifact administration database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropForeignKey(
            name:
"FK_ProductVersionArtifacts_FileArtifacts_FileArtifactId_ArtifactType",
            table: "ProductVersionArtifacts");

        migrationBuilder.DropIndex(
            name: "IX_ProductVersionArtifacts_FileArtifactId_ArtifactType",
            table: "ProductVersionArtifacts");

        migrationBuilder.DropIndex(
            name: "IX_ProductVersionArtifacts_ProductVersionId_ArtifactType",
            table: "ProductVersionArtifacts");

        migrationBuilder.DropUniqueConstraint(
            name: "AK_FileArtifacts_Id_ArtifactType",
            table: "FileArtifacts");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "ProductVersions");

        migrationBuilder.DropColumn(
            name: "ArtifactType",
            table: "ProductVersionArtifacts");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "Products");

        migrationBuilder.AlterColumn<string>(
            name: "Description",
            table: "Products",
            type: "nvarchar(2000)",
            maxLength: 2000,
            nullable: false,
            defaultValue: "",
            oldClrType: typeof(string),
            oldType: "nvarchar(2000)",
            oldMaxLength: 2000,
            oldNullable: true);

        migrationBuilder.CreateIndex(
            name: "IX_ProductVersionArtifacts_FileArtifactId",
            table: "ProductVersionArtifacts",
            column: "FileArtifactId");

        migrationBuilder.AddForeignKey(
            name: "FK_ProductVersionArtifacts_FileArtifacts_FileArtifactId",
            table: "ProductVersionArtifacts",
            column: "FileArtifactId",
            principalTable: "FileArtifacts",
            principalColumn: "Id");
    }
}

```

20260916190522_AddConfigurationRevisionsAndPublishing.cs

```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add configuration revisions and publishing migration for the vendor
    database.
    /// </summary>
    public partial class AddConfigurationRevisionsAndPublishing : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add configuration revisions and publishing database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<Guid>(
                name: "PublishedDefaultRevisionId",
                table: "ProductVersions",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<Guid>(
                name: "PublishedInstallerArtifactId",
                table: "ProductVersions",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<Guid>(
                name: "PublishedSchemaArtifactId",
                table: "ProductVersions",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<Guid>(
                name: "SchemaArtifactId",
                table: "ConfigurationRevisions",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<string>(
                name: "ValidationStatus",
                table: "ConfigurationRevisions",
                type: "nvarchar(32)",
                maxLength: 32,
                nullable: false,
                defaultValue: "LegacyUnverified");

            migrationBuilder.AddColumn<int>(
                name: "LastRevisionNumber",
                table: "ConfigurationProfiles",
                type: "int",
                nullable: false,
```



```

        defaultValue: 0);

migrationBuilder.AddColumn<byte[]>(
    name: "RowVersion",
    table: "ConfigurationProfiles",
    type: "rowversion",
    rowVersion: true,
    nullable: false,
    defaultValue: new byte[0]);

migrationBuilder.CreateTable(
    name: "ConfigurationSourceChanges",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        ConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        ProcessedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ConfigurationSourceChanges", x =>
x.Id);
        table.ForeignKey(
            name:
"FK_ConfigurationSourceChanges_ConfigurationRevisions_ConfigurationRevisionId",
            column: x => x.ConfigurationRevisionId,
            principalTable: "ConfigurationRevisions",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name: "IX_ProductVersions_PublishedDefaultRevisionId",
    table: "ProductVersions",
    column: "PublishedDefaultRevisionId");

migrationBuilder.CreateIndex(
    name: "IX_ProductVersions_PublishedInstallerArtifactId",
    table: "ProductVersions",
    column: "PublishedInstallerArtifactId");

migrationBuilder.CreateIndex(
    name: "IX_ProductVersions_PublishedSchemaArtifactId",
    table: "ProductVersions",
    column: "PublishedSchemaArtifactId");

migrationBuilder.CreateIndex(
    name:
"IX_ConfigurationRevisions_ConfigurationProfileId_ProductVersionId_RevisionNumber",
    table: "ConfigurationRevisions",
    columns: new[] { "ConfigurationProfileId", "ProductVersionId",
"RevisionNumber" });

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationRevisions_SchemaArtifactId",
    table: "ConfigurationRevisions",
    column: "SchemaArtifactId");

```

```

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationSourceChanges_ConfigurationRevisionId",
    table: "ConfigurationSourceChanges",
    column: "ConfigurationRevisionId",
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_ConfigurationSourceChanges_ProcessedAt_CreatedAt",
    table: "ConfigurationSourceChanges",
    columns: new[] { "ProcessedAt", "CreatedAt" });

migrationBuilder.AddForeignKey(
    name: "FK_ConfigurationRevisions_FileArtifacts_SchemaArtifactId",
    table: "ConfigurationRevisions",
    column: "SchemaArtifactId",
    principalTable: "FileArtifacts",
    principalColumn: "Id");

migrationBuilder.AddForeignKey(
    name:
"FK_ProductVersions_ConfigurationRevisions_PublishedDefaultRevisionId",
    table: "ProductVersions",
    column: "PublishedDefaultRevisionId",
    principalTable: "ConfigurationRevisions",
    principalColumn: "Id");

migrationBuilder.AddForeignKey(
    name:
"FK_ProductVersions_FileArtifacts_PublishedInstallerArtifactId",
    table: "ProductVersions",
    column: "PublishedInstallerArtifactId",
    principalTable: "FileArtifacts",
    principalColumn: "Id");

migrationBuilder.AddForeignKey(
    name: "FK_ProductVersions_FileArtifacts_PublishedSchemaArtifactId",
    table: "ProductVersions",
    column: "PublishedSchemaArtifactId",
    principalTable: "FileArtifacts",
    principalColumn: "Id");

migrationBuilder.Sql("UPDATE p SET LastRevisionNumber = COALESCE((SELECT
MAX(r.RevisionNumber) FROM ConfigurationRevisions r WHERE r.ConfigurationProfileId = p.Id), 0)
FROM ConfigurationProfiles p;");

migrationBuilder.Sql("UPDATE v SET PublishedSchemaArtifactId = (SELECT
a.FileArtifactId FROM ProductVersionArtifacts a WHERE a.ProductVersionId = v.Id AND
a.ArtifactType = 'Schema'), PublishedInstallerArtifactId = (SELECT a.FileArtifactId FROM
ProductVersionArtifacts a WHERE a.ProductVersionId = v.Id AND a.ArtifactType = 'Installer') FROM
ProductVersions v WHERE v.Status = 'Published';");

migrationBuilder.Sql("CREATE TRIGGER TR_ConfigurationRevisions_Immutable ON
ConfigurationRevisions AFTER UPDATE, DELETE AS BEGIN SET NOCOUNT ON; THROW 51020, 'Configuration
revision history is immutable.', 1; END;");
}

/// <inheritdoc />
/// <summary>
/// Reverts the add configuration revisions and publishing database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_ConfigurationRevisions_Immutable;");
    migrationBuilder.DropForeignKey(

```

```

        name: "FK_ConfigurationRevisions_FileArtifacts_SchemaArtifactId",
        table: "ConfigurationRevisions");

migrationBuilder.DropForeignKey(
    name:
"FK_ProductVersions_ConfigurationRevisions_PublishedDefaultRevisionId",
    table: "ProductVersions");

migrationBuilder.DropForeignKey(
    name:
"FK_ProductVersions_FileArtifacts_PublishedInstallerArtifactId",
    table: "ProductVersions");

migrationBuilder.DropForeignKey(
    name: "FK_ProductVersions_FileArtifacts_PublishedSchemaArtifactId",
    table: "ProductVersions");

migrationBuilder.DropTable(
    name: "ConfigurationSourceChanges");

migrationBuilder.DropIndex(
    name: "IX_ProductVersions_PublishedDefaultRevisionId",
    table: "ProductVersions");

migrationBuilder.DropIndex(
    name: "IX_ProductVersions_PublishedInstallerArtifactId",
    table: "ProductVersions");

migrationBuilder.DropIndex(
    name: "IX_ProductVersions_PublishedSchemaArtifactId",
    table: "ProductVersions");

migrationBuilder.DropIndex(
    name:
"IX_ConfigurationRevisions_ConfigurationProfileId_ProductVersionId_RevisionNumber",
    table: "ConfigurationRevisions");

migrationBuilder.DropIndex(
    name: "IX_ConfigurationRevisions_SchemaArtifactId",
    table: "ConfigurationRevisions");

migrationBuilder.DropColumn(
    name: "PublishedDefaultRevisionId",
    table: "ProductVersions");

migrationBuilder.DropColumn(
    name: "PublishedInstallerArtifactId",
    table: "ProductVersions");

migrationBuilder.DropColumn(
    name: "PublishedSchemaArtifactId",
    table: "ProductVersions");

migrationBuilder.DropColumn(
    name: "SchemaArtifactId",
    table: "ConfigurationRevisions");

migrationBuilder.DropColumn(
    name: "ValidationStatus",
    table: "ConfigurationRevisions");

migrationBuilder.DropColumn(

```

```

        name: "LastRevisionNumber",
        table: "ConfigurationProfiles");

migrationBuilder.DropColumn(
    name: "RowVersion",
    table: "ConfigurationProfiles");
    }
}
}

```

20260916202147_AddAgentIdentity.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add agent identity migration for the vendor database.
    /// </summary>
    public partial class AddAgentIdentity : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add agent identity database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropIndex(
                name: "IX_Installations_CustomerId",
                table: "Installations");

            migrationBuilder.AddColumn<Guid>(
                name: "AgentId",
                table: "Installations",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<string>(
                name: "InstalledVersion",
                table: "Installations",
                type: "nvarchar(100)",
                maxLength: 100,
                nullable: true);

            migrationBuilder.AddColumn<DateTimeOffset>(
                name: "LastSeenAt",
                table: "Installations",
                type: "datetimeoffset",
                nullable: true);

            migrationBuilder.AddColumn<DateTimeOffset>(
                name: "RegisteredAt",
                table: "Installations",
                type: "datetimeoffset",
                nullable: false,
                defaultValueSql: "SYSDATETIMEOFFSET()");
        }
    }
}

```

```

migrationBuilder.AddColumn<string>(
    name: "Status",
    table: "Installations",
    type: "nvarchar(32)",
    maxLength: 32,
    nullable: false,
    defaultValue: "Active");

migrationBuilder.CreateTable(
    name: "Agents",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        MachineName = table.Column<string>(type: "nvarchar(200)",
maxLength: 200, nullable: false),
        AgentVersion = table.Column<string>(type: "nvarchar(100)",
maxLength: 100, nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        ReportedStatus = table.Column<string>(type: "nvarchar(100)",
maxLength: 100, nullable: true),
        RegisteredAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        LastSeenAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        LastReportedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Agents", x => x.Id);
        table.UniqueConstraint("AK_Agents_Id_CustomerId", x => new {
x.Id, x.CustomerId });

        table.ForeignKey(
            name: "FK_Agents_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "AgentCredentials",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        AgentId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        SecretHash = table.Column<byte[]>(type: "varbinary(32)",
maxLength: 32, nullable: false),
        Purpose = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        ExpiresAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),

```

```

        RevokedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AgentCredentials", x => x.Id);
        table.ForeignKey(
            name: "FK_AgentCredentials_Agents_AgentId",
            column: x => x.AgentId,
            principalTable: "Agents",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "AgentEnrollments",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        CreatedBy = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        SecretHash = table.Column<byte[]>(type: "varbinary(32)",
        Purpose = table.Column<string>(type: "nvarchar(32)",
        CreatedAt = table.Column<DateTimeOffset>(type:
        ExpiresAt = table.Column<DateTimeOffset>(type:
        UsedAt = table.Column<DateTimeOffset>(type:
        RevokedAt = table.Column<DateTimeOffset>(type:
        AgentId = table.Column<Guid>(type: "uniqueidentifier",
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AgentEnrollments", x => x.Id);
        table.ForeignKey(
            name:
"FK_AgentEnrollments_Agents_AgentId_CustomerId",
            columns: x => new { x.AgentId, x.CustomerId },
            principalTable: "Agents",
            principalColumns: new[] { "Id", "CustomerId" });
        table.ForeignKey(
            name: "FK_AgentEnrollments_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "AgentHeartbeats",
    columns: table => new
    {

```

```

        Id = table.Column<long>(type: "bigint", nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        AgentId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        AgentVersion = table.Column<string>(type: "nvarchar(100)",
maxLength: 100, nullable: false),
        Status = table.Column<string>(type: "nvarchar(100)",
maxLength: 100, nullable: false),
        ReportedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        ReceivedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_AgentHeartbeats", x => x.Id);
        table.ForeignKey(
            name: "FK_AgentHeartbeats_Agents_AgentId",
            column: x => x.AgentId,
            principalTable: "Agents",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name: "IX_Installations_AgentId_CustomerId",
    table: "Installations",
    columns: new[] { "AgentId", "CustomerId" });

migrationBuilder.CreateIndex(
    name: "IX_Installations_CustomerId_ProductId_Name",
    table: "Installations",
    columns: new[] { "CustomerId", "ProductId", "Name" },
    unique: true,
    filter: "[AgentId] IS NOT NULL");

migrationBuilder.CreateIndex(
    name: "IX_AgentCredentials_AgentId",
    table: "AgentCredentials",
    column: "AgentId");

migrationBuilder.CreateIndex(
    name: "IX_AgentEnrollments_AgentId_CustomerId",
    table: "AgentEnrollments",
    columns: new[] { "AgentId", "CustomerId" });

migrationBuilder.CreateIndex(
    name: "IX_AgentEnrollments_CustomerId",
    table: "AgentEnrollments",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_AgentHeartbeats_AgentId_ReportedAt",
    table: "AgentHeartbeats",
    columns: new[] { "AgentId", "ReportedAt" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_Agents_CustomerId",
    table: "Agents",
    column: "CustomerId");

migrationBuilder.AddForeignKey(

```

```

        name: "FK_Installations_Agents_AgentId_CustomerId",
        table: "Installations",
        columns: new[] { "AgentId", "CustomerId" },
        principalTable: "Agents",
        principalColumns: new[] { "Id", "CustomerId" });
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add agent identity database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropForeignKey(
            name: "FK_Installations_Agents_AgentId_CustomerId",
            table: "Installations");

        migrationBuilder.DropTable(
            name: "AgentCredentials");

        migrationBuilder.DropTable(
            name: "AgentEnrollments");

        migrationBuilder.DropTable(
            name: "AgentHeartbeats");

        migrationBuilder.DropTable(
            name: "Agents");

        migrationBuilder.DropIndex(
            name: "IX_Installations_AgentId_CustomerId",
            table: "Installations");

        migrationBuilder.DropIndex(
            name: "IX_Installations_CustomerId_ProductId_Name",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "AgentId",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "InstalledVersion",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "LastSeenAt",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "RegisteredAt",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "Status",
            table: "Installations");

        migrationBuilder.CreateIndex(
            name: "IX_Installations_CustomerId",
            table: "Installations",
            column: "CustomerId");
    }

```



```
}  
}
```

20260916204826_AddInstallationLicensing.cs

```
using System;  
using Microsoft.EntityFrameworkCore.Migrations;  
  
#nullable disable  
  
namespace BridgeControl.Data.Migrations.Vendor  
{  
    /// <inheritdoc />  
    /// <summary>  
    /// Defines the add installation licensing migration for the vendor database.  
    /// </summary>  
    public partial class AddInstallationLicensing : Migration  
    {  
        /// <inheritdoc />  
        /// <summary>  
        /// Applies the add installation licensing database migration.  
        /// </summary>  
        protected override void Up(MigrationBuilder migrationBuilder)  
        {  
            migrationBuilder.DropIndex(  
                name: "IX_Installations_CustomerId_ProductId_Name",  
                table: "Installations");  
  
            migrationBuilder.AddColumn<string>(  
                name: "RegistrationName",  
                table: "Installations",  
                type: "nvarchar(200)",  
                maxLength: 200,  
                nullable: true);  
  
            // Freeze the original Agent registration key before allowing display-name  
            migrationBuilder.Sql("UPDATE Installations SET RegistrationName = Name");  
  
            migrationBuilder.AddColumn<byte[]>(  
                name: "RowVersion",  
                table: "Installations",  
                type: "rowversion",  
                rowVersion: true,  
                nullable: false,  
                defaultValue: new byte[0]);  
  
            migrationBuilder.AddUniqueConstraint(  
                name: "AK_Installations_Id_CustomerId_ProductId",  
                table: "Installations",  
                columns: new[] { "Id", "CustomerId", "ProductId" });  
  
            migrationBuilder.CreateTable(  
                name: "Licenses",  
                columns: table => new  
                {  
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable:  
false),  
                    CustomerId = table.Column<Guid>(type: "uniqueidentifier",  
nullable: false),  
                }  
            );  
        }  
    }  
}
```

```

        ProductId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        LicenseReference = table.Column<string>(type:
"nvarchar(200)", maxLength: 200, nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        ValidFrom = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        ValidUntil = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        MaxInstallations = table.Column<int>(type: "int", nullable:
true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Licenses", x => x.Id);

table.UniqueConstraint("AK_Licenses_Id_CustomerId_ProductId", x => new { x.Id, x.CustomerId,
x.ProductId });

        table.CheckConstraint("CK_Licenses_Capacity",
"[MaxInstallations] IS NULL OR [MaxInstallations] > 0");
        table.CheckConstraint("CK_Licenses_Validity", "[ValidUntil]
IS NULL OR [ValidUntil] > [ValidFrom]");
        table.ForeignKey(
            name: "FK_Licenses_Customers_CustomerId",
            column: x => x.CustomerId,
            principalTable: "Customers",
            principalColumn: "Id");
        table.ForeignKey(
            name: "FK_Licenses_Products_ProductId",
            column: x => x.ProductId,
            principalTable: "Products",
            principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "InstallationLicenseAssignments",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        LicenseId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        CustomerId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        ProductId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        AssignedBy = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        AssignedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        RemovedBy = table.Column<Guid>(type: "uniqueidentifier",
nullable: true),
        RemovedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },

```

```

constraints: table =>
{
    table.PrimaryKey("PK_InstallationLicenseAssignments", x =>
x.Id);
    table.ForeignKey(
        name:
"FK_InstallationLicenseAssignments_Installations_InstallationId_CustomerId_ProductId",
        columns: x => new { x.InstallationId, x.CustomerId,
x.ProductId },
        principalTable: "Installations",
        principalColumns: new[] { "Id", "CustomerId",
"ProductId" });
    table.ForeignKey(
        name:
"FK_InstallationLicenseAssignments_Licenses_LicenseId_CustomerId_ProductId",
        columns: x => new { x.LicenseId, x.CustomerId,
x.ProductId },
        principalTable: "Licenses",
        principalColumns: new[] { "Id", "CustomerId",
"ProductId" });
});

migrationBuilder.CreateIndex(
    name: "IX_Installations_CustomerId_ProductId_RegistrationName",
    table: "Installations",
    columns: new[] { "CustomerId", "ProductId", "RegistrationName" },
    unique: true,
    filter: "[AgentId] IS NOT NULL");

migrationBuilder.CreateIndex(
    name: "IX_InstallationLicenseAssignments_InstallationId",
    table: "InstallationLicenseAssignments",
    column: "InstallationId",
    unique: true,
    filter: "[RemovedAt] IS NULL");

migrationBuilder.CreateIndex(
    name:
"IX_InstallationLicenseAssignments_InstallationId_CustomerId_ProductId",
    table: "InstallationLicenseAssignments",
    columns: new[] { "InstallationId", "CustomerId", "ProductId" });

migrationBuilder.CreateIndex(
    name:
"IX_InstallationLicenseAssignments_LicenseId_CustomerId_ProductId",
    table: "InstallationLicenseAssignments",
    columns: new[] { "LicenseId", "CustomerId", "ProductId" });

migrationBuilder.CreateIndex(
    name: "IX_InstallationLicenseAssignments_LicenseId_RemovedAt",
    table: "InstallationLicenseAssignments",
    columns: new[] { "LicenseId", "RemovedAt" });

migrationBuilder.CreateIndex(
    name: "IX_Licenses_CustomerId",
    table: "Licenses",
    column: "CustomerId");

migrationBuilder.CreateIndex(
    name: "IX_Licenses_ProductId",
    table: "Licenses",
    column: "ProductId");

```

```

    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add installation licensing database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "InstallationLicenseAssignments");

        migrationBuilder.DropTable(
            name: "Licenses");

        migrationBuilder.DropUniqueConstraint(
            name: "AK_Installations_Id_CustomerId_ProductId",
            table: "Installations");

        migrationBuilder.DropIndex(
            name: "IX_Installations_CustomerId_ProductId_RegistrationName",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "RegistrationName",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "Installations");

        migrationBuilder.CreateIndex(
            name: "IX_Installations_CustomerId_ProductId_Name",
            table: "Installations",
            columns: new[] { "CustomerId", "ProductId", "Name" },
            unique: true,
            filter: "[AgentId] IS NOT NULL");
    }
}

```

20260916211743_AddEffectiveConfigurationGeneration.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add effective configuration generation migration for the vendor database.
    /// </summary>
    public partial class AddEffectiveConfigurationGeneration : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add effective configuration generation database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)

```

```

{
    migrationBuilder.AddColumn<int>(
        name: "LastEffectiveRevisionNumber",
        table: "Installations",
        type: "int",
        nullable: false,
        defaultValue: 0);

    migrationBuilder.AddColumn<string>(
        name: "InputFingerprint",
        table: "EffectiveConfigurationRevisions",
        type: "nvarchar(64)",
        maxLength: 64,
        nullable: true);

    migrationBuilder.AddColumn<Guid>(
        name: "SchemaArtifactId",
        table: "EffectiveConfigurationRevisions",
        type: "uniqueidentifier",
        nullable: true);

    migrationBuilder.AddColumn<Guid>(
        name: "LastInstallationId",
        table: "ConfigurationSourceChanges",
        type: "uniqueidentifier",
        nullable: true);

    migrationBuilder.CreateTable(
        name: "EffectiveConfigurationSignals",
        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
            ProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
            InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
            LastTargetId = table.Column<Guid>(type: "uniqueidentifier",
nullable: true),
            CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
            ProcessedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_EffectiveConfigurationSignals", x =>
x.Id);

            table.CheckConstraint("CK_EffectiveSignal_Target",
"([InstallationId] IS NULL AND [ProductVersionId] IS NOT NULL) OR ([InstallationId] IS NOT NULL
AND [ProductVersionId] IS NULL)");

            table.ForeignKey(
                name:
"FK_EffectiveConfigurationSignals_Installations_InstallationId",
                column: x => x.InstallationId,
                principalTable: "Installations",
                principalColumn: "Id");

            table.ForeignKey(
                name:
"FK_EffectiveConfigurationSignals_ProductVersions_ProductVersionId",
                column: x => x.ProductVersionId,
                principalTable: "ProductVersions",

```

```

        principalColumn: "Id");
    });

migrationBuilder.CreateTable(
    name: "EffectiveConfigurationWork",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        ProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        RequestedGeneration = table.Column<long>(type: "bigint",
nullable: false),
        CompletedGeneration = table.Column<long>(type: "bigint",
nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        Attempts = table.Column<int>(type: "int", nullable: false),
        NextAttemptAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        UpdatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        LastError = table.Column<string>(type: "nvarchar(2000)",
maxLength: 2000, nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_EffectiveConfigurationWork", x =>
x.Id);
        table.ForeignKey(
            name:
"FK_EffectiveConfigurationWork_Installations_InstallationId",
            column: x => x.InstallationId,
            principalTable: "Installations",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_EffectiveConfigurationWork_ProductVersions_ProductVersionId",
            column: x => x.ProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
    });

migrationBuilder.CreateIndex(
    name:
"IX_EffectiveConfigurationRevisions_InstallationId_InputFingerprint",
    table: "EffectiveConfigurationRevisions",
    columns: new[] { "InstallationId", "InputFingerprint" },
    unique: true,
    filter: "[InputFingerprint] IS NOT NULL");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationRevisions_SchemaArtifactId",
    table: "EffectiveConfigurationRevisions",
    column: "SchemaArtifactId");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationSignals_InstallationId",

```

```

        table: "EffectiveConfigurationSignals",
        column: "InstallationId");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationSignals_ProcessedAt_CreatedAt",
    table: "EffectiveConfigurationSignals",
    columns: new[] { "ProcessedAt", "CreatedAt" });

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationSignals_ProductVersionId",
    table: "EffectiveConfigurationSignals",
    column: "ProductVersionId");

migrationBuilder.CreateIndex(
    name:
"IX_EffectiveConfigurationWork_InstallationId_ProductVersionId",
    table: "EffectiveConfigurationWork",
    columns: new[] { "InstallationId", "ProductVersionId" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationWork_ProductVersionId",
    table: "EffectiveConfigurationWork",
    column: "ProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_EffectiveConfigurationWork_Status_NextAttemptAt",
    table: "EffectiveConfigurationWork",
    columns: new[] { "Status", "NextAttemptAt" });

migrationBuilder.AddForeignKey(
    name:
"FK_EffectiveConfigurationRevisions_FileArtifacts_SchemaArtifactId",
    table: "EffectiveConfigurationRevisions",
    column: "SchemaArtifactId",
    principalTable: "FileArtifacts",
    principalColumn: "Id");

migrationBuilder.Sql("UPDATE i SET LastEffectiveRevisionNumber =
COALESCE((SELECT MAX(r.RevisionNumber) FROM EffectiveConfigurationRevisions r WHERE
r.InstallationId = i.Id), 0) FROM Installations i;");
migrationBuilder.Sql("INSERT INTO EffectiveConfigurationSignals (Id,
InstallationId, CreatedAt) SELECT NEWID(), Id, SYSDATETIMEOFFSET() FROM Installations;");
migrationBuilder.Sql("CREATE TRIGGER
TR_EffectiveConfigurationRevisions_Immutable ON EffectiveConfigurationRevisions AFTER UPDATE,
DELETE AS BEGIN SET NOCOUNT ON; THROW 51030, 'Effective configuration history is immutable.', 1;
END;");

migrationBuilder.Sql("CREATE TRIGGER
TR_EffectiveConfigurationSources_Immutable ON EffectiveConfigurationSourceRevisions AFTER UPDATE,
DELETE AS BEGIN SET NOCOUNT ON; THROW 51031, 'Effective configuration lineage is immutable.', 1;
END;");
}

/// <inheritdoc />
/// <summary>
/// Reverts the add effective configuration generation database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_EffectiveConfigurationSources_Immutable;");
    migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_EffectiveConfigurationRevisions_Immutable;");
}

```

```

        migrationBuilder.DropForeignKey(
            name:
"FK_EffectiveConfigurationRevisions_FileArtifacts_SchemaArtifactId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropTable(
            name: "EffectiveConfigurationSignals");

        migrationBuilder.DropTable(
            name: "EffectiveConfigurationWork");

        migrationBuilder.DropIndex(
            name:
"IX_EffectiveConfigurationRevisions_InstallationId_InputFingerprint",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropIndex(
            name: "IX_EffectiveConfigurationRevisions_SchemaArtifactId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropColumn(
            name: "LastEffectiveRevisionNumber",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "InputFingerprint",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropColumn(
            name: "SchemaArtifactId",
            table: "EffectiveConfigurationRevisions");

        migrationBuilder.DropColumn(
            name: "LastInstallationId",
            table: "ConfigurationSourceChanges");
    }
}

```

20260917061100_AddConfigurationMigrationReviews.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add configuration migration reviews migration for the vendor database.
    /// </summary>
    public partial class AddConfigurationMigrationReviews : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add configuration migration reviews database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {

```



```

migrationBuilder.AddColumn<int>(
    name: "RevisionNumber",
    table: "ConfigurationMigrations",
    type: "int",
    nullable: false,
    defaultValue: 0);

migrationBuilder.AddColumn<byte[]>(
    name: "RowVersion",
    table: "ConfigurationMigrations",
    type: "rowversion",
    rowVersion: true,
    nullable: false,
    defaultValue: new byte[0]);

migrationBuilder.CreateTable(
    name: "ConfigurationMigrationReviews",
    columns: table => new
    {
        Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
        UserId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
        InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        TargetProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        Fingerprint = table.Column<string>(type: "nvarchar(64)",
maxLength: 64, nullable: false),
        PlanJson = table.Column<string>(type: "nvarchar(max)",
nullable: false),
        CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        AppliedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        ResultRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
        ApprovedRuleIdsJson = table.Column<string>(type:
"nvarchar(max)", nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_ConfigurationMigrationReviews", x =>
x.Id);

        table.ForeignKey(
            name:
"FK_ConfigurationMigrationReviews_EffectiveConfigurationRevisions_ResultRevisionId",
            column: x => x.ResultRevisionId,
            principalTable: "EffectiveConfigurationRevisions",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_ConfigurationMigrationReviews_Installations_InstallationId",
            column: x => x.InstallationId,
            principalTable: "Installations",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_ConfigurationMigrationReviews_ProductVersions_TargetProductVersionId",
            column: x => x.TargetProductVersionId,

```

```

                principalTable: "ProductVersions",
                principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_ConfigurationMigrationReviews_UserAccounts_UserId",
            column: x => x.UserId,
            principalTable: "UserAccounts",
            principalColumn: "Id");
    });

    migrationBuilder.CreateTable(
        name: "MigrationDefinitionRevisions",
        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
            MigrationId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
            RevisionNumber = table.Column<int>(type: "int", nullable:
false),
            RulesJson = table.Column<string>(type: "nvarchar(max)",
nullable: false),
            CreatedByUserId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
            CreatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_MigrationDefinitionRevisions", x =>
x.Id);

            table.ForeignKey(
                name:
"FK_MigrationDefinitionRevisions_ConfigurationMigrations_MigrationId",
                column: x => x.MigrationId,
                principalTable: "ConfigurationMigrations",
                principalColumn: "Id");

            table.ForeignKey(
                name:
"FK_MigrationDefinitionRevisions_UserAccounts_CreatedByUserId",
                column: x => x.CreatedByUserId,
                principalTable: "UserAccounts",
                principalColumn: "Id");
        });

    migrationBuilder.CreateIndex(
        name: "IX_ConfigurationMigrationReviews_InstallationId",
        table: "ConfigurationMigrationReviews",
        column: "InstallationId");

    migrationBuilder.CreateIndex(
        name: "IX_ConfigurationMigrationReviews_ResultRevisionId",
        table: "ConfigurationMigrationReviews",
        column: "ResultRevisionId");

    migrationBuilder.CreateIndex(
        name: "IX_ConfigurationMigrationReviews_TargetProductVersionId",
        table: "ConfigurationMigrationReviews",
        column: "TargetProductVersionId");

    migrationBuilder.CreateIndex(

```

```

        name:
"IX_ConfigurationMigrationReviews_UserId_InstallationId_TargetProductVersionId_CreatedAt",
        table: "ConfigurationMigrationReviews",
        columns: new[] { "UserId", "InstallationId",
"TargetProductVersionId", "CreatedAt" });

migrationBuilder.CreateIndex(
    name: "IX_MigrationDefinitionRevisions_CreatedByUserId",
    table: "MigrationDefinitionRevisions",
    column: "CreatedByUserId");

migrationBuilder.CreateIndex(
    name: "IX_MigrationDefinitionRevisions_MigrationId_RevisionNumber",
    table: "MigrationDefinitionRevisions",
    columns: new[] { "MigrationId", "RevisionNumber" },
    unique: true);

migrationBuilder.Sql("CREATE TRIGGER
TR_MigrationDefinitionRevisions_Immutable ON MigrationDefinitionRevisions AFTER UPDATE, DELETE AS
BEGIN SET NOCOUNT ON; THROW 51040, 'Migration definition history is immutable.', 1; END;");

migrationBuilder.Sql("CREATE TRIGGER
TR_ConfigurationMigrationReviews_Immutable ON ConfigurationMigrationReviews AFTER UPDATE, DELETE
AS BEGIN SET NOCOUNT ON; IF NOT EXISTS (SELECT 1 FROM inserted) OR UPDATE(Id) OR UPDATE(UserId)
OR UPDATE(InstallationId) OR UPDATE(TargetProductVersionId) OR UPDATE(Fingerprint) OR
UPDATE(PlanJson) OR UPDATE(CreatedAt) OR EXISTS (SELECT 1 FROM deleted WHERE AppliedAt IS NOT
NULL) THROW 51041, 'Migration review evidence is immutable.', 1; END;");
    }

    /// <inheritdoc />
    /// <summary>
    /// Reverts the add configuration migration reviews database migration.
    /// </summary>
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_ConfigurationMigrationReviews_Immutable;");
        migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_MigrationDefinitionRevisions_Immutable;");
        migrationBuilder.DropTable(
            name: "ConfigurationMigrationReviews");

        migrationBuilder.DropTable(
            name: "MigrationDefinitionRevisions");

        migrationBuilder.DropColumn(
            name: "RevisionNumber",
            table: "ConfigurationMigrations");

        migrationBuilder.DropColumn(
            name: "RowVersion",
            table: "ConfigurationMigrations");
    }
}

```

20260917074602_AddInstallationStateReconciliation.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

```

```

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add installation state reconciliation migration for the vendor database.
    /// </summary>
    public partial class AddInstallationStateReconciliation : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add installation state reconciliation database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<Guid>(
                name: "DesiredStateRevisionId",
                table: "Installations",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.AddColumn<long>(
                name: "LastDesiredStateRevisionNumber",
                table: "Installations",
                type: "bigint",
                nullable: false,
                defaultValue: 0L);

            migrationBuilder.AddColumn<Guid>(
                name: "ReportedStateReportId",
                table: "Installations",
                type: "uniqueidentifier",
                nullable: true);

            migrationBuilder.CreateTable(
                name: "DesiredStateRevisions",
                columns: table => new
                {
                    Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
                    InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
                    ProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
                    EffectiveConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
                    RevisionNumber = table.Column<long>(type: "bigint",
nullable: false),
                    CreatedByUserId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
                    UpdatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_DesiredStateRevisions", x => x.Id);

                    table.UniqueConstraint("AK_DesiredStateRevisions_Id_InstallationId", x => new { x.Id,
x.InstallationId });

                    table.ForeignKey(

```

```

name:
"FK_DesiredStateRevisions_EffectiveConfigurationRevisions_EffectiveConfigurationRevisionId_InstallationId",
columns: x => new {
x.EffectiveConfigurationRevisionId, x.InstallationId },
principalTable: "EffectiveConfigurationRevisions",
principalColumns: new[] { "Id", "InstallationId" });
table.ForeignKey(
name:
"FK_DesiredStateRevisions_Installations_InstallationId",
column: x => x.InstallationId,
principalTable: "Installations",
principalColumn: "Id");
table.ForeignKey(
name:
"FK_DesiredStateRevisions_ProductVersions_ProductVersionId",
column: x => x.ProductVersionId,
principalTable: "ProductVersions",
principalColumn: "Id");
table.ForeignKey(
name:
"FK_DesiredStateRevisions_UserAccounts_CreatedByUserId",
column: x => x.CreatedByUserId,
principalTable: "UserAccounts",
principalColumn: "Id");
});

migrationBuilder.CreateTable(
name: "InstallationStateReports",
columns: table => new
{
Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
AgentId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
InstalledProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
EffectiveConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
ConfigurationHash = table.Column<string>(type:
"nvarchar(64)", maxLength: 64, nullable: true),
Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
ReportedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
ReceivedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false)
},
constraints: table =>
{
table.PrimaryKey("PK_InstallationStateReports", x => x.Id);

table.UniqueConstraint("AK_InstallationStateReports_Id_InstallationId", x => new { x.Id,
x.InstallationId });

table.ForeignKey(
name: "FK_InstallationStateReports_Agents_AgentId",
column: x => x.AgentId,
principalTable: "Agents",
principalColumn: "Id");
table.ForeignKey(

```

```

name:
"FK_InstallationStateReports_EffectiveConfigurationRevisions_EffectiveConfigurationRevisionId_InstallationId",
columns: x => new {
x.EffectiveConfigurationRevisionId, x.InstallationId },
principalTable: "EffectiveConfigurationRevisions",
principalColumns: new[] { "Id", "InstallationId" });
table.ForeignKey(
name:
"FK_InstallationStateReports_Installations_InstallationId",
column: x => x.InstallationId,
principalTable: "Installations",
principalColumn: "Id");
table.ForeignKey(
name:
"FK_InstallationStateReports_ProductVersions_InstalledProductVersionId",
column: x => x.InstalledProductVersionId,
principalTable: "ProductVersions",
principalColumn: "Id");
});

migrationBuilder.CreateTable(
name: "StateReconciliationWork",
columns: table => new
{
InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
RequestedGeneration = table.Column<long>(type: "bigint",
nullable: false),
CompletedGeneration = table.Column<long>(type: "bigint",
nullable: false),
Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
Reason = table.Column<string>(type: "nvarchar(100)",
maxLength: 100, nullable: true),
Attempts = table.Column<int>(type: "int", nullable: false),
NextAttemptAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
UpdatedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
},
constraints: table =>
{
table.PrimaryKey("PK_StateReconciliationWork", x =>
x.InstallationId);
table.ForeignKey(
name:
"FK_StateReconciliationWork_Installations_InstallationId",
column: x => x.InstallationId,
principalTable: "Installations",
principalColumn: "Id");
});

migrationBuilder.CreateTable(
name: "Deployments",
columns: table => new
{
Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),

```

```

        InstallationId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        DesiredStateRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        TargetProductVersionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        TargetConfigurationRevisionId = table.Column<Guid>(type:
"uniqueidentifier", nullable: false),
        ReconciliationGeneration = table.Column<long>(type:
"bigint", nullable: false),
        OperationType = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        Status = table.Column<string>(type: "nvarchar(32)",
maxLength: 32, nullable: false),
        IsActive = table.Column<bool>(type: "bit", nullable: false),
        ReconciledReportId = table.Column<Guid>(type:
"uniqueidentifier", nullable: true),
        RequestedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
        CompletedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: true),
        RowVersion = table.Column<byte[]>(type: "rowversion",
rowVersion: true, nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_Deployments", x => x.Id);
        table.ForeignKey(
            name:
"FK_Deployments_DesiredStateRevisions_DesiredStateRevisionId_InstallationId",
            columns: x => new { x.DesiredStateRevisionId,
x.InstallationId },
            principalTable: "DesiredStateRevisions",
            principalColumns: new[] { "Id", "InstallationId" });
        table.ForeignKey(
            name:
"FK_Deployments_EffectiveConfigurationRevisions_TargetConfigurationRevisionId_InstallationId",
            columns: x => new { x.TargetConfigurationRevisionId,
x.InstallationId },
            principalTable: "EffectiveConfigurationRevisions",
            principalColumns: new[] { "Id", "InstallationId" });
        table.ForeignKey(
            name:
"FK_Deployments_InstallationStateReports_ReconciledReportId_InstallationId",
            columns: x => new { x.ReconciledReportId,
x.InstallationId },
            principalTable: "InstallationStateReports",
            principalColumns: new[] { "Id", "InstallationId" });
        table.ForeignKey(
            name: "FK_Deployments_Installations_InstallationId",
            column: x => x.InstallationId,
            principalTable: "Installations",
            principalColumn: "Id");
        table.ForeignKey(
            name:
"FK_Deployments_ProductVersions_TargetProductVersionId",
            column: x => x.TargetProductVersionId,
            principalTable: "ProductVersions",
            principalColumn: "Id");
    });

```

```

migrationBuilder.CreateTable(

```

```

        name: "DeploymentAttempts",
        columns: table => new
        {
            Id = table.Column<Guid>(type: "uniqueidentifier", nullable:
false),
            DeploymentId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
            AttemptNumber = table.Column<int>(type: "int", nullable:
false),
            Status = table.Column<string>(type: "nvarchar(32)",
            ErrorCode = table.Column<string>(type: "nvarchar(100)",
            ErrorMessage = table.Column<string>(type: "nvarchar(2000)",
            StartedAt = table.Column<DateTimeOffset>(type:
            CompletedAt = table.Column<DateTimeOffset>(type:
            RowVersion = table.Column<byte[]>(type: "rowversion",
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_DeploymentAttempts", x => x.Id);
            table.ForeignKey(
                name:
"FK_DeploymentAttempts_Deployments_DeploymentId",
                column: x => x.DeploymentId,
                principalTable: "Deployments",
                principalColumn: "Id");
        });

migrationBuilder.CreateIndex(
    name: "IX_Installations_DesiredStateRevisionId_Id",
    table: "Installations",
    columns: new[] { "DesiredStateRevisionId", "Id" });

migrationBuilder.CreateIndex(
    name: "IX_Installations_ReportedStateReportId_Id",
    table: "Installations",
    columns: new[] { "ReportedStateReportId", "Id" });

migrationBuilder.CreateIndex(
    name: "IX_DeploymentAttempts_DeploymentId_AttemptNumber",
    table: "DeploymentAttempts",
    columns: new[] { "DeploymentId", "AttemptNumber" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_Deployments_DesiredStateRevisionId_InstallationId",
    table: "Deployments",
    columns: new[] { "DesiredStateRevisionId", "InstallationId" });

migrationBuilder.CreateIndex(
    name: "IX_Deployments_InstallationId",
    table: "Deployments",
    column: "InstallationId",
    unique: true,
    filter: "[IsActive] = 1");

migrationBuilder.CreateIndex(

```



```

        name: "IX_Deployments_InstallationId_ReconciliationGeneration",
        table: "Deployments",
        columns: new[] { "InstallationId", "ReconciliationGeneration" },
        unique: true);

migrationBuilder.CreateIndex(
    name: "IX_Deployments_ReconciledReportId_InstallationId",
    table: "Deployments",
    columns: new[] { "ReconciledReportId", "InstallationId" });

migrationBuilder.CreateIndex(
    name:
"IX_Deployments_TargetConfigurationRevisionId_InstallationId",
    table: "Deployments",
    columns: new[] { "TargetConfigurationRevisionId", "InstallationId"
});

migrationBuilder.CreateIndex(
    name: "IX_Deployments_TargetProductVersionId",
    table: "Deployments",
    column: "TargetProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_DesiredStateRevisions_CreatedByUserId",
    table: "DesiredStateRevisions",
    column: "CreatedByUserId");

migrationBuilder.CreateIndex(
    name:
"IX_DesiredStateRevisions_EffectiveConfigurationRevisionId_InstallationId",
    table: "DesiredStateRevisions",
    columns: new[] { "EffectiveConfigurationRevisionId",
"InstallationId" });

migrationBuilder.CreateIndex(
    name: "IX_DesiredStateRevisions_InstallationId_RevisionNumber",
    table: "DesiredStateRevisions",
    columns: new[] { "InstallationId", "RevisionNumber" },
    unique: true);

migrationBuilder.CreateIndex(
    name: "IX_DesiredStateRevisions_ProductVersionId",
    table: "DesiredStateRevisions",
    column: "ProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_InstallationStateReports_AgentId",
    table: "InstallationStateReports",
    column: "AgentId");

migrationBuilder.CreateIndex(
    name:
"IX_InstallationStateReports_EffectiveConfigurationRevisionId_InstallationId",
    table: "InstallationStateReports",
    columns: new[] { "EffectiveConfigurationRevisionId",
"InstallationId" });

migrationBuilder.CreateIndex(
    name: "IX_InstallationStateReports_InstallationId_ReportedAt",
    table: "InstallationStateReports",
    columns: new[] { "InstallationId", "ReportedAt" },
    unique: true);

```

```

migrationBuilder.CreateIndex(
    name: "IX_InstallationStateReports_InstalledProductVersionId",
    table: "InstallationStateReports",
    column: "InstalledProductVersionId");

migrationBuilder.CreateIndex(
    name: "IX_StateReconciliationWork_Status_NextAttemptAt",
    table: "StateReconciliationWork",
    columns: new[] { "Status", "NextAttemptAt" });

migrationBuilder.AddForeignKey(
    name:
"FK_Installations_DesiredStateRevisions_DesiredStateRevisionId_Id",
    table: "Installations",
    columns: new[] { "DesiredStateRevisionId", "Id" },
    principalTable: "DesiredStateRevisions",
    principalColumns: new[] { "Id", "InstallationId" });

migrationBuilder.AddForeignKey(
    name:
"FK_Installations_InstallationStateReports_ReportedStateReportId_Id",
    table: "Installations",
    columns: new[] { "ReportedStateReportId", "Id" },
    principalTable: "InstallationStateReports",
    principalColumns: new[] { "Id", "InstallationId" });
migrationBuilder.Sql("CREATE TRIGGER TR_DesiredStateRevisions_Immutable ON
DesiredStateRevisions AFTER UPDATE, DELETE AS BEGIN SET NOCOUNT ON; THROW 51050, 'Desired state
history is immutable.', 1; END;");
migrationBuilder.Sql("CREATE TRIGGER TR_InstallationStateReports_Immutable
ON InstallationStateReports AFTER UPDATE, DELETE AS BEGIN SET NOCOUNT ON; THROW 51051, 'Accepted
reported state history is immutable.', 1; END;");
}

/// <inheritdoc />
/// <summary>
/// Reverts the add installation state reconciliation database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_DesiredStateRevisions_Immutable;");
    migrationBuilder.Sql("DROP TRIGGER IF EXISTS
TR_InstallationStateReports_Immutable;");
    migrationBuilder.DropForeignKey(
        name:
"FK_Installations_DesiredStateRevisions_DesiredStateRevisionId_Id",
        table: "Installations");

    migrationBuilder.DropForeignKey(
        name:
"FK_Installations_InstallationStateReports_ReportedStateReportId_Id",
        table: "Installations");

    migrationBuilder.DropTable(
        name: "DeploymentAttempts");

    migrationBuilder.DropTable(
        name: "StateReconciliationWork");

    migrationBuilder.DropTable(
        name: "Deployments");

```

```

        migrationBuilder.DropTable(
            name: "DesiredStateRevisions");

        migrationBuilder.DropTable(
            name: "InstallationStateReports");

        migrationBuilder.DropIndex(
            name: "IX_Installations_DesiredStateRevisionId_Id",
            table: "Installations");

        migrationBuilder.DropIndex(
            name: "IX_Installations_ReportedStateReportId_Id",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "DesiredStateRevisionId",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "LastDesiredStateRevisionNumber",
            table: "Installations");

        migrationBuilder.DropColumn(
            name: "ReportedStateReportId",
            table: "Installations");
    }
}

```

20260917081112_AddDeploymentExecution.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add deployment execution migration for the vendor database.
    /// </summary>
    public partial class AddDeploymentExecution : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add deployment execution database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<DateTimeOffset>(name: "ResultReceivedAt", table:
"DeploymentAttempts", type: "datetimeoffset", nullable: true);
            migrationBuilder.DropForeignKey(
                name: "FK_DeploymentAttempts_Deployments_DeploymentId",
                table: "DeploymentAttempts");

            migrationBuilder.AddColumn<Guid>(
                name: "AgentId",
                table: "DeploymentAttempts",

```

```

        type: "uniqueidentifier",
        nullable: false,
        defaultValue: new Guid("00000000-0000-0000-0000-000000000000"));

migrationBuilder.AddColumn<Guid>(
    name: "InstallationId",
    table: "DeploymentAttempts",
    type: "uniqueidentifier",
    nullable: false,
    defaultValue: new Guid("00000000-0000-0000-0000-000000000000"));

migrationBuilder.AddColumn<String>(
    name: "LastReportHash",
    table: "DeploymentAttempts",
    type: "nvarchar(64)",
    maxLength: 64,
    nullable: true);

migrationBuilder.AddUniqueConstraint(
    name: "AK_Deployments_Id_InstallationId",
    table: "Deployments",
    columns: new[] { "Id", "InstallationId" });

// Sprint 13 did not dispatch attempts. Preserve any existing imported rows
// where ownership is known.
migrationBuilder.Sql("UPDATE a SET InstallationId = d.InstallationId,
AgentId = i.AgentId FROM DeploymentAttempts a JOIN Deployments d ON d.Id = a.DeploymentId JOIN
Installations i ON i.Id = d.InstallationId;");

migrationBuilder.CreateIndex(
    name: "IX_DeploymentAttempts_AgentId",
    table: "DeploymentAttempts",
    column: "AgentId");

migrationBuilder.CreateIndex(
    name: "IX_DeploymentAttempts_DeploymentId_InstallationId",
    table: "DeploymentAttempts",
    columns: new[] { "DeploymentId", "InstallationId" });

migrationBuilder.CreateIndex(
    name: "IX_DeploymentAttempts_InstallationId",
    table: "DeploymentAttempts",
    column: "InstallationId",
    unique: true,
    filter: "[CompletedAt] IS NULL");

migrationBuilder.AddForeignKey(
    name: "FK_DeploymentAttempts_Agents_AgentId",
    table: "DeploymentAttempts",
    column: "AgentId",
    principalTable: "Agents",
    principalColumn: "Id");

migrationBuilder.AddForeignKey(
    name:
"FK_DeploymentAttempts_Deployments_DeploymentId_InstallationId",
    table: "DeploymentAttempts",
    columns: new[] { "DeploymentId", "InstallationId" },
    principalTable: "Deployments",
    principalColumns: new[] { "Id", "InstallationId" });
migrationBuilder.Sql("")

```

```

        CREATE TRIGGER TR_DeploymentAttempts_ImmutableCompleted ON DeploymentAttempts
AFTER UPDATE, DELETE AS
    BEGIN
        SET NOCOUNT ON;
        IF EXISTS (SELECT 1 FROM deleted WHERE CompletedAt IS NOT NULL)
            THROW 51000, 'Completed deployment attempts are immutable.', 1;
        IF EXISTS (SELECT 1 FROM deleted d LEFT JOIN inserted i ON i.Id = d.Id
            WHERE i.Id IS NULL OR i.DeploymentId <> d.DeploymentId OR
i.InstallationId <> d.InstallationId
                OR i.AgentId <> d.AgentId OR i.AttemptNumber <> d.AttemptNumber OR
i.StartedAt <> d.StartedAt)
            THROW 51000, 'Deployment attempt identity is immutable.', 1;
        END
    """);
}

/// <inheritdoc />
/// <summary>
/// Reverts the add deployment execution database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP TRIGGER
TR_DeploymentAttempts_ImmutableCompleted;");
    migrationBuilder.DropColumn(name: "ResultReceivedAt", table:
"DeploymentAttempts");
    migrationBuilder.DropForeignKey(
        name: "FK_DeploymentAttempts_Agents_AgentId",
        table: "DeploymentAttempts");

    migrationBuilder.DropForeignKey(
        name:
"FK_DeploymentAttempts_Deployments_DeploymentId_InstallationId",
        table: "DeploymentAttempts");

    migrationBuilder.DropUniqueConstraint(
        name: "AK_Deployments_Id_InstallationId",
        table: "Deployments");

    migrationBuilder.DropIndex(
        name: "IX_DeploymentAttempts_AgentId",
        table: "DeploymentAttempts");

    migrationBuilder.DropIndex(
        name: "IX_DeploymentAttempts_DeploymentId_InstallationId",
        table: "DeploymentAttempts");

    migrationBuilder.DropIndex(
        name: "IX_DeploymentAttempts_InstallationId",
        table: "DeploymentAttempts");

    migrationBuilder.DropColumn(
        name: "AgentId",
        table: "DeploymentAttempts");

    migrationBuilder.DropColumn(
        name: "InstallationId",
        table: "DeploymentAttempts");

    migrationBuilder.DropColumn(
        name: "LastReportHash",
        table: "DeploymentAttempts");

```

```

        migrationBuilder.AddForeignKey(
            name: "FK_DeploymentAttempts_Deployments_DeploymentId",
            table: "DeploymentAttempts",
            column: "DeploymentId",
            principalTable: "Deployments",
            principalColumn: "Id");
    }
}
}

```

20260917090823_AddOperationalHistory.cs

```

using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add operational history migration for the vendor database.
    /// </summary>
    public partial class AddOperationalHistory : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add operational history database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "LogIngestionWindows",
                columns: table => new
                {
                    AgentId = table.Column<Guid>(type: "uniqueidentifier",
nullable: false),
                    StartedAt = table.Column<DateTimeOffset>(type:
"datetimeoffset", nullable: false),
                    Batches = table.Column<int>(type: "int", nullable: false),
                    Entries = table.Column<int>(type: "int", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_LogIngestionWindows", x => x.AgentId);
                    table.ForeignKey(
                        name: "FK_LogIngestionWindows_Agents_AgentId",
                        column: x => x.AgentId,
                        principalTable: "Agents",
                        principalColumn: "Id");
                });

            migrationBuilder.CreateTable(
                name: "OperationalLogs",
                columns: table => new
                {
                    Id = table.Column<long>(type: "bigint", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),

```

```

        nullable: false),
        AgentId = table.Column<Guid>(type: "uniqueidentifier",
        "uniqueidentifier", nullable: true),
        InstallationId = table.Column<Guid>(type:
        Severity = table.Column<string>(type: "nvarchar(16)",
        Source = table.Column<string>(type: "nvarchar(100)",
        Message = table.Column<string>(type: "nvarchar(2000)",
        IsSensitive = table.Column<bool>(type: "bit", nullable:
        Fingerprint = table.Column<string>(type: "nvarchar(64)",
        CreatedAt = table.Column<DateTimeOffset>(type:
        ReceivedAt = table.Column<DateTimeOffset>(type:
        "datetimeoffset", nullable: false),
        "datetimeoffset", nullable: false)
    },
    constraints: table =>
    {
        table.PrimaryKey("PK_OperationalLogs", x => x.Id);
        table.ForeignKey(
            name: "FK_OperationalLogs_Agents_AgentId",
            column: x => x.AgentId,
            principalTable: "Agents",
            principalColumn: "Id");
        table.ForeignKey(
            name:
            "FK_OperationalLogs_Installations_InstallationId",
            column: x => x.InstallationId,
            principalTable: "Installations",
            principalColumn: "Id");
    });

    migrationBuilder.CreateIndex(
        name: "IX_VendorAuditLogs_CreatedAt_Id",
        table: "VendorAuditLogs",
        columns: new[] { "CreatedAt", "Id" });

    migrationBuilder.CreateIndex(
        name: "IX_OperationalLogs_AgentId_CreatedAt_Id",
        table: "OperationalLogs",
        columns: new[] { "AgentId", "CreatedAt", "Id" });

    migrationBuilder.CreateIndex(
        name: "IX_OperationalLogs_AgentId_Fingerprint",
        table: "OperationalLogs",
        columns: new[] { "AgentId", "Fingerprint" },
        unique: true);

    migrationBuilder.CreateIndex(
        name: "IX_OperationalLogs_InstallationId_CreatedAt_Id",
        table: "OperationalLogs",
        columns: new[] { "InstallationId", "CreatedAt", "Id" });

    migrationBuilder.CreateIndex(
        name: "IX_OperationalLogs_ReceivedAt",
        table: "OperationalLogs",
        column: "ReceivedAt");
}

```

```

        /// <inheritdoc />
        /// <summary>
        /// Reverts the add operational history database migration.
        /// </summary>
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "LogIngestionWindows");

            migrationBuilder.DropTable(
                name: "OperationalLogs");

            migrationBuilder.DropIndex(
                name: "IX_VendorAuditLogs_CreatedAt_Id",
                table: "VendorAuditLogs");
        }
    }
}

```

20260918121548_AddApiManagedApplications.cs

```

using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace BridgeControl.Data.Migrations.Vendor
{
    /// <inheritdoc />
    /// <summary>
    /// Defines the add API managed applications migration for the vendor database.
    /// </summary>
    public partial class AddApiManagedApplications : Migration
    {
        /// <inheritdoc />
        /// <summary>
        /// Applies the add API managed applications database migration.
        /// </summary>
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.AddColumn<string>(
                name: "DeploymentSettingsJson",
                table: "ProductVersions",
                type: "nvarchar(max)",
                nullable: true);

            migrationBuilder.AddColumn<string>(
                name: "DeploymentSettingsJson",
                table: "Products",
                type: "nvarchar(max)",
                nullable: true);

            migrationBuilder.AddColumn<string>(
                name: "InstallerIdentityJson",
                table: "FileArtifacts",
                type: "nvarchar(max)",
                nullable: true);
        }

        /// <inheritdoc />
        /// <summary>

```



```
/// Reverts the add API managed applications database migration.
/// </summary>
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropColumn(
        name: "DeploymentSettingsJson",
        table: "ProductVersions");

    migrationBuilder.DropColumn(
        name: "DeploymentSettingsJson",
        table: "Products");

    migrationBuilder.DropColumn(
        name: "InstallerIdentityJson",
        table: "FileArtifacts");
}
}
```